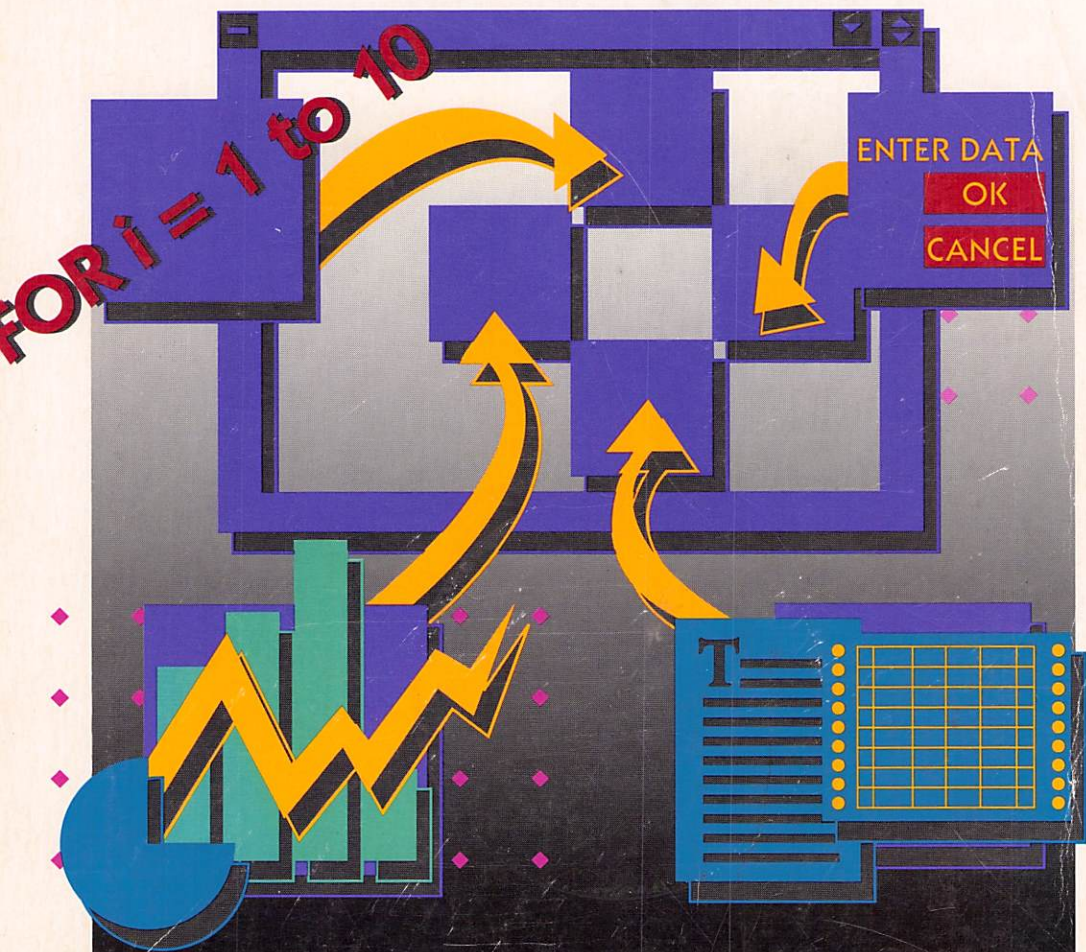


CA-REALIZER[®]



Programming Guide

For Microsoft Windows And IBM OS/2

**COMPUTER[®]
ASSOCIATES**
Software superior by design.

CA-Realizer™

For Microsoft® Windows™ and IBM® OS/2®

Version 2.0

Programming Guide

September 1993

© Copyright 1991-1993 Computer Associates International, Inc. All rights reserved.
Printed in the United States of America

Computer Associates International, Inc.
Publisher

No part of this documentation may be copied, photocopied, reproduced, translated, microfilmed, or otherwise duplicated on any medium without written consent of Computer Associates International, Inc.

Use of the software programs described herein and this documentation is subject to the Computer Associates License Agreement enclosed in the software package.

All product names referenced herein are trademarks of their respective companies.

Contents

Chapter 1 : Introduction

Overview	1-1
How to Use This Guide	1-2
What You Need to Know	1-6
Working with Windows and OS/2	1-6
Conventions Used in This Guide	1-7
Standard Conventions	1-7
Notational Conventions	1-9
Getting Help	1-10

Chapter 2 : Variables and Data Types

In This Chapter	2-1
CA-Realizer Data Types	2-1
Simple vs. Complex Data Types	2-2
Simple Data Types	2-5
Numeric Data Types	2-5
String Data Types	2-8
Date-Time Data Types	2-9
Complex Data Types	2-10
Array Data Types	2-10
Records	2-11
Family Data Types	2-11
Nested Complex Data Structures	2-12
Variables and Constants	2-13
Variables	2-14
Constants	2-20

Data Typing in CA-Realizer	2-21
Implicit vs. Explicit Data Typing	2-21
The DIM Statement	2-22
Creating User-defined Data Types	2-23
Using Arrays.....	2-25
The General Form of an Array	2-26
Constructing Arrays	2-28
Array-handling Features Unique to CA-Realizer	2-29
Operating on Arrays	2-38
Arrays of Arrays (Subarrays)	2-40
Limits on Sizes of Arrays	2-42
Printing Arrays	2-44
Creating Record Data Types	2-45
Records in Nested Complex Data Structures	2-49
Creating and Using Families.....	2-51
Creating a Family	2-52
Referencing Members of a Family	2-52
Manipulating Families and Members	2-53
Arrays of Families	2-53
Families in Nested Complex Data Structures	2-54

Chapter 3 : Manipulating Numbers

In This Chapter	3-1
Working with Operators, Expressions, and Functions.....	3-1
Operators	3-2
Expressions	3-9
Functions	3-10
Working with Array Values	3-11
Working with Mathematical Values	3-15
Trigonometric and Exponential Functions	3-15
Rounding Functions	3-17
Random Functions	3-19
Array-Specific Functions	3-20
Reporting Mathematical Errors	3-30

Working with Date-Time Values	3-30
Creating and Converting to Date-Time Values	3-31
Setting the System Date and Time	3-33
Examining Date-Time Values	3-33
Manipulating Date-Time Values	3-35

Chapter 4 : Manipulating Strings

In This Chapter	4-1
CA-Realizer String Basics	4-1
Converting Between Text and ASCII Codes	4-2
Combining Strings	4-4
Comparing Strings	4-4
Retrieving Parts of a String	4-6
Retrieving from the Left, Right, or Middle of a String	4-6
Retrieving from Any Position in the String	4-8
Replacing Text Within Strings	4-12
Changing Case	4-14
Inserting and Deleting Portions of a String	4-15
Converting Strings	4-16
Numbers to Strings	4-16
Strings to Numbers and Dates	4-17
Converting from External Formats	4-19
Formatting Strings	4-20
The Sprint Function	4-20
Format Codes	4-21

Chapter 5 : Controlling the Flow of a Program

In This Chapter	5-1
Forming Statements	5-1
Control Structures	5-2
Conditional Statements	5-4
The IF Statement	5-4
The IF Function	5-7

The SELECT Statement	5-8
Exiting IF and SELECT Statements	5-9
Unconditionally Changing Flow with GOTO	5-9
Iterative Statements	5-10
The FOR...NEXT Statement	5-11
The LOOP Statement	5-13
The WHILE Statement	5-13
Exiting an Iterative Statement	5-14
Optimizing Iteration by Using Arrays	5-15
Nested Control Structures	5-17
Trapping Errors with ON ERROR GOTO	5-18
The ON ERROR GOTO Statement	5-18
Writing a Simple Error Handler	5-19
Disabling Error-Trapping	5-21

Chapter 6 : Structured Programming with Procedures and Functions

In This Chapter	6-1
Using Procedures and Functions	6-1
Determining When to Use a Procedure or Function	6-2
Defining a Procedure or Function	6-2
Using Parameters in Procedures and Functions	6-8
Scoping Variables in Procedures and Functions	6-14
Using Modifiers in Procedures and Functions	6-16
Including Optional Parameters and Modifiers	6-17
Redefining Procedures and Functions	6-21
Functions and Expression Evaluation	6-21
Recursion	6-22
Creating and Running CA-Realizer Modules	6-23
Creating Libraries	6-23
Using Modules as Procedures	6-23
Setting the Hourglass Cursor	6-24
Using the EXECUTE Command	6-24

Chapter 7 : File I/O

In This Chapter	7-1
File Input and Output in CA-Realizer	7-1
Available File Commands and Functions	7-3
Basic File Concepts	7-4
Using the Low-Level File Manipulation Commands and Functions	7-11
Opening a File	7-11
Writing to a File	7-12
Closing a File	7-15
Reading a File	7-15
Manipulating the File Pointer	7-20
Using the High-Level File Manipulation Commands and Functions	7-23
Importing/Exporting Data	7-24
Storing and Retrieving Sets of Variables	7-24
File Formats	7-26
Importing and Exporting CA-Realizer Named Data	7-28
Importing and Exporting CA-Realizer Plain Data	7-29
Importing and Exporting Spreadsheet Named Data	7-30
Importing and Exporting Spreadsheet Plain Data	7-31
Importing and Exporting Text Named Data	7-31
Importing and Exporting Text Plain Data	7-33
Binary Files	7-34
Importing and Exporting XBase Data Files	7-37

Chapter 8 : Controlling Input and Output

In This Chapter	8-1
The Standard Dialog Boxes	8-1
Selecting a File to Open	8-2
Changing the Printer Configuration	8-6
Changing a Font	8-8
Changing Colors	8-10
The Input Box	8-11
Displaying Messages Using Message Boxes	8-14

Printing	8-16
Printing Arrays	8-17
Printing to Logs	8-17
Formatting Output	8-18
Sending Output to a Printer	8-30

Chapter 9 : Overview of the CA-Realizer Programmable Application Tools

In This Chapter	9-1
Creating Tools	9-2
Assigning Tool IDs	9-4
Creating a Tool	9-6
Selecting and Querying a Tool	9-10
Controlling a Tool	9-12
Displaying a Tool	9-13
Sizing and Positioning a Tool	9-14
Printing a Tool	9-19
Closing a Tool	9-20
Setting Tool Procedures	9-21
Specifying a Tool Procedure	9-21
Writing Asynchronous Tool Procedures	9-30
The Application Procedure	9-32
A Tool Example	9-33
Tool Command Summary	9-35
Animator Commands and Functions	9-35
Board Commands and Functions	9-35
Chart Commands and Functions	9-35
Form Commands and Functions	9-36
Graphics Tablet Commands and Functions	9-37
Log Commands and Functions	9-37
Menu Commands and Functions	9-38
Scheduler Commands and Functions	9-38
Spreadsheet Commands and Functions	9-38

Chapter 10 : Using Fonts and Pictures in Tools

In This Chapter	10-1
Fonts	10-2
Creating a Font	10-2
Selecting, Querying, and Closing a Font	10-4
Pictures	10-6
Creating a Picture	10-6
Selecting, Querying, and Closing a Picture	10-9
Using the Clipboard	10-10
A Picture Example	10-11

Chapter 11 : Charts

In This Chapter	11-1
The Anatomy of a Chart	11-4
Creating a New Chart	11-5
Creating the Chart	11-5
Setting the Chart Type and Displaying the Chart	11-6
Available Plot Types	11-7
Controlling the Appearance of a Chart	11-19
Setting Chart Colors	11-19
Setting Line Styles	11-24
Setting Chart Axes	11-26
Adding a Chart Key	11-33
Labeling Charts	11-37
Controlling the Chart Grid	11-43
Setting Fonts for a Chart	11-45
Advanced Charting Features	11-48
Updating a Plot	11-48
Removing a Plot	11-49
Querying Axis Values	11-49
Creating Multiple Panes	11-50
Controlling the Redraw	11-52

Setting a Chart Procedure	11-53
Attaching a Chart Procedure	11-53
Processing in the Chart Procedure	11-54
Converting Screen-Relative to Data-Relative Positions	11-54
A Sample Chart Procedure	11-55
Printing a Chart	11-57

Chapter 12 : Spreadsheets

In This Chapter	12-1
The Anatomy of a Spreadsheet	12-2
Creating a New Spreadsheet	12-3
Using SheetNew	12-3
Spreadsheet Data	12-4
Controlling a Spreadsheet	12-6
Setting Fonts for Cells and Headings	12-8
Using Column Buttons	12-10
Restricting Cells, Rows, and Columns	12-11
Setting the Default Format	12-12
Moving to a Cell	12-13
Controlling Row and Column Headers	12-14
Changing Row Headers	12-14
Changing Column Headers	12-15
Hiding Row and Column Headers	12-16
Updating the Contents of a Spreadsheet	12-17
Adding New Variables	12-17
Recalculating the Data	12-18
Removing Cleared Data	12-18
Example	12-19
Querying the Spreadsheet Selection	12-20
Setting a Spreadsheet Procedure	12-22
Attaching a Spreadsheet Procedure	12-22
Processing in the Spreadsheet Procedure	12-23
A Sample Spreadsheet Procedure	12-24

Chapter 13 : Boards

In This Chapter	13-1
Displaying Data in a Board	13-2
Creating a Board	13-3
Specifying a Board Template	13-5
Updating the Contents of a Board	13-10
Adding More Data	13-10
Recalculating the Data	13-12
Removing Cleared Data	13-12
Setting a Board Procedure	13-13
Attaching a Board Procedure	13-13
Processing in the Board Procedure	13-14
A Sample Board Procedure	13-15

Chapter 14 : Text Logs

In This Chapter	14-1
Creating a Log	14-2
Displaying Information in a Log	14-3
Controlling the Appearance of a Log	14-5
Clearing a Log	14-5
Setting a Font	14-6
Setting Tab Stops	14-7
Obtaining Information About a Log and Its Contents	14-8
Determining the Number of Characters	14-9
Extracting Text From a Log	14-9
Extracting a Line of Text From a Log	14-9
Manipulating the Information in a Log	14-10
Setting a Log Procedure	14-12
Attaching a Log Procedure	14-12
Processing in the Log Procedure	14-13
A Sample Log Procedure	14-13

Chapter 15 : Forms

In This Chapter	15-1
What Is a Form?	15-1
Form Commands and Functions	15-2
Form Objects	15-3
Buttons	15-3
Option Buttons	15-4
Option Bitmaps	15-4
Check Boxes	15-4
Check Bitmaps	15-5
Group Boxes	15-5
Edit Controls	15-5
Captions	15-6
Bitmaps, Metafiles, and PCX Pictures	15-6
List Boxes	15-7
Frames	15-8
Scroll Bars	15-8
Digital Clocks	15-9
Event Timers	15-9
OLE Objects	15-9
CA-Realizer Tools	15-9
Creating a New Form	15-10
Controlling a Form	15-14
Obtaining Information About a Form	15-17
Setting the Colors in a Form	15-18
Setting the Background Color	15-19
Setting the Color of an Object	15-19
Adding Form Objects	15-20
Using the FormSetObject Command	15-21
Adding Buttons	15-25
Adding Option Buttons	15-25
Adding Bitmap Option Buttons	15-26
Adding Check Boxes	15-27
Adding Bitmap Check Boxes	15-27
Adding Group Boxes	15-28

Adding Edit Controls	15-28
Adding Captions	15-29
Adding Pictures	15-30
Adding List Boxes	15-31
Combining Modifier Groups	15-34
Adding Frames	15-34
Adding Scroll Bars	15-34
Adding Digital Clocks	15-36
Adding Event Timers	15-36
Adding OLE Objects	15-37
Adding CA-Realizer Tools	15-37
Positioning Objects in a Form	15-38
Manipulating and Modifying Form Objects	15-39
Closing an Object	15-41
Changing the State of an Object	15-41
Changing the Color of an Existing Object	15-42
Setting the Focus to an Object	15-42
Adjusting the Front-to-Back and Tab-Stop Ordering of Objects	15-43
Changing the Status of an Option Button or Check Box	15-44
Modifying a List Box	15-44
Modifying a Scroll Bar	15-45
Modifying an Event Timer	15-45
Querying the Object	15-46
Querying the Object Value	15-47
Obtaining the Numeric Value of an Object with FormQNum	15-47
Obtaining the Text Value of an Object with FormQStr	15-48
Creating Toolbars, Status Bars, and Palettes Using Non-MDI Forms	15-50
Non-MDI Forms	15-50
Adjusting the MDI Region	15-51
Creating a Toolbar or Status Bar	15-51
Creating a Palette	15-52
Creating a Pop-up Form	15-53
Form Processing	15-54
Notification Messages	15-55

Modal Forms	15-59
Edit Control Verification	15-59
Modal Pick and PickDrag	15-61
An Example of a Modal Form.....	15-62
Modeless Forms	15-65
Form Procedures	15-66
The Structure of the Form Procedure	15-66
Modeless Form Messages	15-67
Edit Control Verification	15-69
Modeless Pick and PickDrag.....	15-69
An Example of a Modeless Form.....	15-70
Setting a Background Grid	15-72
Adding Accelerator Keys	15-73
Drag and Drop	15-75
Setting Draggable Objects and Drop Receivers	15-76
Processing Drag and Drop Messages	15-77
A Drag and Drop Example	15-77

Chapter 16 : Graphics Tablets

In This Chapter	16-1
Creating Images	16-1
Creating a Graphics Tablet	16-2
Selecting a Tablet	16-3
Drawing on a Tablet	16-4
Pens and Brushes	16-5
Drawing Lines	16-7
Drawing Simple Shapes	16-9
Drawing Polygons	16-12
Drawing Bitmaps and Individual Pixels	16-13
Returning the Color of an Individual Pixel	16-14
Drawing Text	16-15
Trapping Mouse Clicks in a Tablet	16-16

Buffering Modes	16-17
Metafile Mode	16-17
BufferedMetafile Mode	16-17
Bitmap Mode	16-17
Controlling the Redraw	16-18
Undoing Commands	16-20
A Tablet in Action—Tic Tac Toe	16-21
A Tablet in Action—Simple Draw!	16-23

Chapter 17 : The Animator

In This Chapter	17-1
How the Animator Works	17-2
Creating an Animation	17-3
Defining Cells	17-4
Defining Frames	17-6
Controlling the Animation	17-7
Starting and Stopping the Animation	17-7
Controlling the Animation Speed	17-8
Controlling the Animation Position	17-8
Setting Special Frames	17-8
Notification of Mouse Clicks	17-9
Using Pictures with Animation	17-10

Chapter 18 : Custom Menus and Accelerator Keys

In This Chapter	18-1
The Anatomy of a Menu	18-1
Creating a Menu	18-3
Generating an ID	18-3
Specifying an ID	18-4
Creating the Menu	18-4
Specifying a Keyboard Accelerator	18-4
Selecting a Menu	18-5

Controlling a Menu	18-6
Displaying a Menu	18-7
Controlling Menu Redraw	18-9
Repositioning Menus	18-9
Closing a Menu	18-9
Adding Items to a Menu	18-10
Custom Menu Commands	18-10
Separators	18-11
Modifying Menu Items	18-12
Setting a Menu Procedure	18-14
Attaching a Menu Procedure	18-14
A Sample Menu Procedure	18-15
Using CA-Realizer Menu Commands in Your Menus	18-17
Adding Submenus to a Menu	18-18
Creating a Submenu	18-19
Adding Items to a Submenu	18-20
Using Accelerators	18-21

Chapter 19 : The Scheduler

In This Chapter	19-1
The Anatomy of the Scheduler	19-2
Invoking the Scheduler	19-3
How Scheduled Commands are Run	19-4
Scheduling an Event	19-4
Repeating an Event	19-7
Starting and Stopping the Scheduler	19-8
Removing an Event from the Scheduler	19-9
Querying the Scheduler and Its Contents	19-10

Chapter 20 : Help Files

In This Chapter	20-1
Using the System Help Manager	20-1
Using a Help Procedure	20-3

Chapter 21 : CA-Realizer System Variables

In This Chapter	21-1
SetSys and QSys Parameters	21-2
The Size Variables	21-5
The Path Variables	21-7
The Log Variables	21-9
Default PRINT Formats	21-10
Command-Specific System Variables	21-11
System Constants	21-13

Chapter 22 : Accessing Operating System Commands

In This Chapter	22-1
Launching Other Applications	22-1
Running Operating System Commands	22-3
Creating Batch Files with CA-Realizer	22-4
How Windows Commands Synchronize with CA-Realizer	22-5
Setting the System Date and Time	22-6

Chapter 23 : The Clipboard

In This Chapter	23-1
Manipulating Text	23-1
Copying Text to the Clipboard	23-1
Retrieving Text from the Clipboard	23-2
Manipulating Pictures	23-2
Placing a Picture on the Clipboard	23-2
Retrieving a Picture from the Clipboard	23-2

Chapter 24 : Serial Communications

In This Chapter	24-1
Opening a Serial Port	24-2
Reading from a Serial Port	24-3
Writing to a Serial Port	24-4
Querying a Serial Port	24-5
Closing a Serial Port	24-6

Chapter 25 : Dynamic Data Exchange and CA-RET

In This Chapter	25-1
What Is DDE?	25-1
How DDE Works	25-2
Initiating a Client DDE Session	25-3
Selecting a DDE Session	25-5
DDE Data Formats	25-5
Windows Data Formats	25-6
OS/2 Data Formats	25-7
Sending Messages to a Server	25-8
Responding to Messages from a Server	25-9
Waiting for a DDE Message	25-10
Ending a DDE Session	25-11
A Sample Client Session	25-11
Using CA-Realizer as a DDE Server	25-13

Responding to Messages from a Client	25-16
Obtaining Information about a DDE Session	25-17
Using CA-RET with CA-REALIZER	25-18
Initiating a Conversation with CA-RET	25-18
Executing a CA-RET Command	25-18
Requesting Information from CA-RET	25-19
Terminating the CA-RET Conversation	25-19
A Sample CA-RET Conversation	25-19

Chapter 26 : External Procedures and Functions

In This Chapter	26-1
Dynamic Link Libraries	26-2
Declaring External Procedures	26-3
Passing Numeric Parameters	26-4
Passing Character Parameters	26-6
Passing Structure Parameters	26-7
Passing Pointer Parameters	26-9
Declaring External Functions	26-10
Accessing System DLL Functions	26-12
ODBC Interface Library	26-13

Chapter 27 : Custom Controls

In This Chapter	27-1
What Is a Custom Control?	27-2
Declaring Custom Controls	27-3
Placing a Custom Control in a Form	27-4
Adding Custom Controls to FormDev	27-5
Creating Custom Control Libraries	27-5
Defining Constants	27-5
Registering Custom Controls	27-6
Custom Control Options	27-8
Creating a Custom Control DLL	27-10
Windows and OS/2 Messages	27-10

CA-Realizer Notification Messages	27-12
CA-Realizer Query Messages	27-12
CA-Realizer Numeric Value Messages	27-14
CA-Realizer String Value Messages	27-15
Order of Messages	27-17
Custom Messages	27-18
Sending Messages to CA-Realizer	27-19
Posting a Message to the Form Procedure	27-19
Sending a Message to the Form Procedure Immediately	27-20
Style Flags	27-20
Form Colors	27-21

Chapter 28 : Encrypting CA-Realizer Programs

In This Chapter	28-1
Saving Encrypted Files	28-1
The Scope of Encrypted Symbols	28-2
Exporting Symbols	28-2
Local Symbols	28-4
Clearing Exported Symbols	28-5

Appendix A : CA-Realizer Keywords

Appendix B : CA-Realizer Predefined Constants

Appendix C : The ASCII Character Set

Index

Chapter 1

Introduction

Overview

This guide provides a guided tour through the fundamentals of programming using the CA-Realizer BASIC language. Topics include:

- Data types
- Manipulation of numeric and date-time values, strings, and files
- Control of program flow
- Structured programming with procedures and functions (including External procedures and functions)
- File and data input and output
- The CA-Realizer programmable application tools
- Serial communications
- DDE and DLLs
- Custom controls

Most of the topics presented in this guide use sample programs and code fragments. To better understand the examples provided, it would be most beneficial to enter the sample code in CA-Realizer as each is presented.

Note: This guide discusses how to use most of the commands provided by the CA-Realizer language. If you need additional detailed technical information, please refer to the *CA-Realizer Language Reference Guide*.

How to Use This Guide

This guide is organized into the following chapters.

1. Introduction

2. Variables and Data Types

Introduces the data types supported by CA-Realizer and the various data manipulation elements that comprise a CA-Realizer program.

3. Manipulating Numbers

Describes how to use variables, operators, and functions to form mathematical expressions.

4. Manipulating Strings

Describes how to manipulate strings using the many CA-Realizer string manipulation commands and functions.

5. Controlling the Flow of a Program

Instructs you in forming statements for controlling the flow of your CA-Realizer programs. These statements allow you to alter the ordinary top-to-bottom program flow by enabling the program to react to data values and user input.

6. Structured Programming with Procedures and Functions

Discusses how to define blocks of statements—that are repeated throughout a program—as procedures and functions.

7. File I/O

Details how to read, write, and save files in CA-Realizer. Basic file manipulation concepts, file formats, and CA-Realizer I/O commands and functions are also discussed.

8. Controlling Input and Output

Presents the CA-Realizer standard dialog box functions for controlling file input and output.

9. Overview of the CA-Realizer Programmable Application Tools

Introduces the CA-Realizer programmable application tools including: spreadsheets, charts, text logs, boardwatches, graphics tablets, menus, The Animator, and The Scheduler.

10. Using Fonts and Pictures in Tools

Provides information related to using fonts and pictures to enhance your CA-Realizer programs.

11. Charts

Instructs you in the use of the chart tool for displaying data. The various CA-Realizer chart related commands and functions are also discussed.

12. Spreadsheets

Discusses the spreadsheet tool and the associated commands and functions that allow you to view and manipulate CA-Realizer variables.

13. Boards

Details how to use, control, and program boardwatches. The various CA-Realizer board related commands and functions are also presented.

14. Text Logs

Describes how to use and control the text log tool using the log-related CA-Realizer commands and functions.

15. Forms

Introduces you to the use of forms for creating appealing and flexible user interfaces. Information related to creating, modifying, and processing forms including the available CA-Realizer form commands, functions, form objects, and form types is also presented.

16. Graphics Tablets

Details how to use a graphics tablet for drawings and plots. The CA-Realizer commands and functions that allow you to create graphics and control their content and format are also presented.

17. The Animator

Introduces the CA-Realizer Animator that allows you to create animation sequences in your programs.

18. Custom Menus and Accelerator Keys

Describes how to create, use, add, and remove menus and accelerator keys in your CA-Realizer programs.

19. The Scheduler

Discusses how you can use the CA-Realizer Scheduler to run programs and commands at specified times and intervals.

20. Help Files

Details how to create and display online help information using the Help Compiler and Help Manager.

21. CA-Realizer System Variables

Presents the variables that can be accessed to query and fine-tune the CA-Realizer system.

22. Accessing Operating System Commands

Describes how to launch other applications, run the CA-Realizer operating system commands, create batch files, and synchronize Windows and CA-Realizer commands.

23. The Clipboard

Discusses the use of the Clipboard for reading and writing text and pictures.

24. Serial Communications

Details how to open, read from, write to, query, and close a serial port.

25. Dynamic Data Exchange and CA-RET

Explains what Dynamic Data Exchange (DDE) is and how it can be used. Using the Computer Associates visual report generator, CA-RET, is also discussed.

26. External Procedures and Functions

Discusses the use of Dynamic Link Libraries (DLLs) and how to declare external procedures and functions. The Open Database Connectivity (ODBC) interface is also presented.

27. Custom Controls

Describes how to create and use custom controls in your CA-Realizer programs.

28. Encrypting CA-Realizer Programs

Details how to encrypt your CA-Realizer programs, limiting user access to source code, procedures, or variables.

Appendix A: CA-Realizer Keywords

Lists the CA-Realizer Version 2.0 keywords.

Appendix B: CA-Realizer Predefined Constants

Lists the CA-Realizer Version 2.0 predefined constants.

Appendix C: The ASCII Character Set

Presents the ASCII character set used by CA-Realizer Version 2.0.

Index

What You Need to Know

This guide assumes you are familiar with Microsoft Windows and OS/2 navigational techniques and terminology, and command and mouse conventions. It is also assumed that you are familiar with terms such as object, icon, click, double-click, drag, select, choose, menu, window, dialog box, and Clipboard.

If you are new to Windows, OS/2, or graphical user interfaces (GUIs), be sure to read your Microsoft Windows or IBM OS/2 documentation.

Working with Windows and OS/2

You can install and use this product under Microsoft Windows or IBM OS/2, or with Windows installed under OS/2. Once you are using the product itself, these operating environments make such a small difference that a single set of CA-Realizer guides is sufficient.

Some of the screens, windows, and dialog boxes in these guides were created under Windows, others under OS/2. There are small cosmetic differences, but most windows have the same content under either operating environment. For example, the system menu icon in the upper-left corner of an active window looks different—in Windows it displays a horizontal bar, while in OS/2 it shows the product icon—but the menu choices that are available when you click the button are the same.

In cases where functional differences in the Windows and OS/2 versions do exist, each version is shown.

Conventions Used in This Guide

Standard Conventions

Key Combinations Whenever two keys are joined together with a plus (+) sign (for example, CTRL+R), you should hold down the first key while pressing the second key to complete the command. Release the second key in this kind of combination first.

Key Sequences When keys are separated by a comma (,), press them in the sequence indicated. The keystroke command ALT+E,C, for example, indicates that you should hold the ALT key down while pressing the E key, release them both, and then press and release the C key.

Key Names The names of keys are spelled out in the document as they appear on your keyboard (where possible)—for example, ENTER, CTRL, and DEL.

Note that when referring to the four arrow keys on your keyboard as a group, they are referred to as Direction keys; however, the name of each Direction key is used when referring to them individually.

Cross References The following conventions are used:

- Guide name in italics:
See the *CA-Realizer Language Reference Guide*.
- Chapter name in double quotes:
See "The Project Builder" in the *CA-Realizer User Guide*.
- Section name as it appears in the document:
Also see the Linking Forms Together section.

MENU COMMAND KEY NAME	This typeface indicates the name of a menu command, button, or key.
Program Example	This typeface shows sample programs or program fragments. Lines written in this font are examples you should follow. Type this literal information into CA-Realizer exactly as shown.
Program Output	This typeface shows output from sample programs.
<i>term</i>	Italics introduce a CA-Realizer term for the first time. Italics with a command indicate a user-defined variable.
[]	Brackets appear around optional items in a CA-Realizer command definition.
...	Ellipses are used in the definition of a CA-Realizer command to indicate that you can repeat a parameter.
	This image, when present in the picture of a window, indicates that the user should click the mouse button.
	This image, when present in the picture of a window, indicates that the user should drag the mouse while holding down the mouse button.
	This image indicates that the information following it is specific to Microsoft Windows.
	This image indicates that the information following it is specific to IBM OS/2.

Notational Conventions

When a function that returns a value is presented, a number of symbols are displayed to the left of the equal sign in the syntax. These symbols are used to indicate the data type of the value that is returned.

First Character

The first character in the prefix indicates the *type* the parameter should be given as:

Character	Represents
<i>n</i>	Numeric data type (integer or real)
<i>s</i>	String data type
<i>d</i>	Date-time data type
<i>g</i>	General data type (any type)

Second Character

The second character in the prefix indicates whether the parameter must be an array or a scalar:

Character	Represents
<i>s</i>	Scalar
<i>a</i>	Array (any number of dimensions)
<i>v</i>	Vector (one-dimensional array)
<i>m</i>	Matrix (two-dimensional array of numeric values)
<i>g</i>	General (scalar or array)

For example, the *ns* in the following syntax indicates that the MenuQ function returns a numeric scalar:

```
ns = MenuQ(Action [; MenuNum])
```

Getting Help



CA-Realizer provides Online Help, which can be used to display information on your console as you work. By choosing the commands on the Help menu or pressing the F1 key, you can receive help about CA-Realizer menus, dialog boxes, commands and functions, and keyboard shortcuts.

You can get context-sensitive help for a menu, menu command, or dialog box by pressing the F1 key when the desired menu or menu command is selected, or when the desired dialog box is displayed.

In addition, when a program file is open in a Program window, you can receive context-sensitive help for the commands and functions in the open program file. Simply place the insertion point cursor in the command/function for which you want help and press the F1 key. CA-Realizer automatically displays the syntax for the selected command/function. (If there is no recognizable command at the location of the insertion point, the main help index appears.)

Note: To receive online help for Help, press the F1 key from anywhere within the CA-Realizer Help system, or select USING HELP from the HELP menu.

Chapter 2

Variables and Data Types

In This Chapter	2-1
CA-Realizer Data Types	2-1
Simple vs. Complex Data Types	2-3
Numeric Data Types	2-4
Integers	2-5
Reals	2-6
String Data Types	2-8
Date-Time Data Types	2-9
Array Data Types	2-9
Family Data Types	2-10
Variables and Constants	2-10
Variables	2-10
Assigning a Value to a Variable	2-11
Valid Variable Names	2-12
Variable Declarations	2-13
Changing a Variable's Type	2-14
Obtaining Information About a Variable	2-15
Clearing Variables	2-16
Exchanging Values	2-16
Constants	2-17
Symbolic Constants	2-17
Predefined Constants	2-17
Data Typing in CA-Realizer	2-18
Implicit vs. Explicit Data Typing	2-18
The DIM Statement	2-20
Creating User-defined Data Types	2-22
Creating Record Data Types	2-24
Defining a Record	2-25

Declaring a Record Variable	2-26
Accessing the Fields of a Record	2-26
Arrays of Records	2-26
Extracting a Column from an Array of Records	2-27
Using Arrays	2-28
The General Form of an Array	2-28
Constructing Arrays	2-30
Array-handling Features Unique to CA-Realizer	2-32
Array Subranging	2-32
Dynamic Array Expansion	2-35
Changing the Number of Dimensions of an Array	2-37
Operating on Arrays	2-38
Arrays of Arrays (Subarrays)	2-40
Limits on Sizes of Arrays	2-42
Printing Arrays	2-43
Using Families	2-44
Creating a Family	2-44
Referencing Members of a Family	2-44
Manipulating Families and Members	2-45
Arrays of Families	2-45

Chapter 2

Variables and Data Types

In This Chapter

This chapter discusses the data types supported by CA-Realizer, as well as some of the various data manipulation elements that can be used in your programs. These elements include: constants, variables, arrays, records and families. CA-Realizer provides a powerful set of capabilities, including 11 simple data types and 4 complex data types, that permit you to efficiently create and utilize extremely general data structures with great power and simplicity.

CA-Realizer Data Types

CA-Realizer supports a wide range of data types for use as variables and components of arrays, records, or families. The supported data types can be grouped into two categories—*simple* and *complex*—which are introduced below and discussed in greater detail later in this chapter.

Simple vs. Complex Data Types

CA-Realizer supports 11 distinct simple data types. You can declare any variable to be one of these types by using a DIM statement. These data types can also be used to define the components of the complex data types described below.

Simple Data Types

Like most versions of the BASIC language, CA-Realizer supports the use of two of the most fundamental simple data types—*numerics* and *strings*. Strings are also known as *alphas*, and both terms are used interchangeably throughout this guide. There are 8 numeric data types, 6 of which are for integer numbers while 2 are for real numbers. There are also 2 string data types.

CA-Realizer also supports a third basic data type—*date-time*, which provides important additional capabilities for manipulating date and/or time information. The 11 simple data types are listed below.

Simple Data Type	Description
BYTE	A 1-byte unsigned integer
WORD	A 2-byte unsigned integer
DWORD	A 4-byte unsigned integer
CHAR	A 1-byte signed integer
INTEGER	A 2-byte signed integer
LONG	A 4-byte signed integer

Continued

Continued

Simple Data Type	Description
SINGLE	A 4-byte floating point number
DOUBLE	An 8-byte floating point number
STRING	A variable-length string
STRING * <i>n</i>	A fixed-length string
DATETIME	An 8-byte date-time value

Of the 11 simple data types, there are 4 default data types that are used in cases where the data type is not explicitly declared. They are the STRING data type, the DATETIME data type, the LONG data type for integers, and the DOUBLE data type for real numbers.

Complex Data Types

There are also 4 complex data types that allow a program to manipulate and store related data values using a single variable. CA-Realizer supports several advanced data types, including fixed arrays, dynamic arrays, records, and families. These complex data types allows one or more components to be related by a single array, record, or family name. The components are called elements in the case of arrays, fields in the case of records, and members in the case of families.

Each of the 4 complex data types allows you to reference a specific component, whether it is a simple data value or another complex data type, by using the name of the array, record, or family AND additional information that defines the component. In the case of arrays, one or more index numbers are required to specify one of the elements in the array. In the case of records or families, a field or member name is also required.

These complex data types provide powerful capabilities since the components may be simple data types, or may even be other complex data types. The power in using the complex data types arises from the fact that general multi-level data structures involving nested complex variables can be created. For example, you can have arrays of records or arrays of families. Or, you can have a family that contains arrays, records, and other families as components.

The table below describes the 4 complex data types. Additional information about them will be presented later in this chapter.

Complex Data Type	Description
ARRAY (<i>Dims</i>) AS <i>Type</i>	A fixed array of elements with the same data type
ARRAY	A dynamic array of elements with the same data type
RECORD	A fixed collection of fields of different data types
FAMILY	A dynamic collection of members of different data types

Fixed arrays are fixed in terms of the data type of the elements, as well as the number of elements. In dynamic arrays the data type of the elements is fixed, but the number of elements can be changed “on the fly”. Records have a fixed set of fields and field types, while families allow for changes in the data type of a member as well as for members to be created or destroyed “on the fly”.

Note: The RECORD data type is different than the other data types listed above in that it cannot be used in DIM statements. RECORD is not a keyword like the names of the other data types. Instead the specific name of the defined record structure must be used.

User Defined Types

You can also define any of the above data types as a user-defined type, by assigning it a different name in a TYPE statement. Instead of declaring several variables to be INTEGER types, you could define MYTYPE1 to be INTEGER, and then use a DIM statement to declare a variable with data type MYTYPE1. Later, if you wish to change all of them to LONG, you need only change the single TYPE statement that defines MYTYPE1.

Simple Data Types

Numeric Data Types

Numeric data types fall into two classes—*integers* and *real numbers*. CA-Realizer supports several different data types for each class, with different valid ranges of data values, as summarized below:

Data Type	Description	Valid Range of Values
BYTE	1-byte unsigned integer	0 to 255
WORD	2-byte unsigned integer	0 to 65,535
DWORD	4-byte unsigned integer	0 to 4,294,967,295
CHAR	1-byte signed integer	-128 to 127

Continued

Continued

Data Type	Description	Valid Range of Values
INTEGER	2-byte signed integer	-32,768 to 32,767
LONG	4-byte signed integer	-2,147,483,648 to 2,147,483,647
SINGLE	4-byte floating point value	Approximately 1.4E-45 to 3.4E+38 for either positive or negative numbers
DOUBLE	8-byte floating point value	Approximately 4.94E-324 to 1.8E+308 for either positive or negative numbers

Integers

Integers are positive or negative *whole numbers*. The advantage of these numbers is that they always represent an integer value with perfect precision. This precision is important in certain types of programs and advanced operations, such as when passing numerics to Dynamic Link Libraries (DLLs), saving numbers to records, or directly accessing the Windows or OS/2 APIs.

There are 6 integer data types, that require 1, 2, or 4 bytes of storage for a single value. By default, the LONG data type is used if the data type of an integer variable has not been declared. Not only does CA-Realizer support a wide range of integer types, but it is also quite flexible because it allows you to specify values for these types using various notations.

For example, you can specify integer values in binary, octal, or hexadecimal notation by prefixing them with &B, &O, or &H, respectively (for example, &B11010 (26 decimal), &O177 (255 decimal), or &H8C9D (35997 decimal)). Hexadecimal numbers can include the digits from 0 to 9, as well as either uppercase or lowercase letters from A to F.

Reals

Real values are numbers like 1.0 or 3.141593. CA-Realizer supports two types of real data types—*single-precision* and *double-precision*. They differ in the number of decimal places to which the values they store are accurate, as well as in the minimum and maximum allowed values. See the preceding table for the minimum and maximum values.

Single-precision numbers are accurate up to 7 decimal places, while double-precision numbers are accurate to 15 places (accuracy is ultimately determined by floating point characteristics of your computer's floating point hardware or software emulator). You can declare a variable to be either a SINGLE or DOUBLE data type by using the DIM statement. By default, CA-Realizer stores reals as double-precision floating point values if the data type of a numeric variable has not been declared.

You enter real numbers by typing in the values using either normal or exponential notation. Exponential notation uses a format called *E notation*, wherein the value following the E is a power of ten. For example, the value 3,500,000 can be written as 3.5E+6, and small numbers, such as -0.001257, can be written as -1.257E-3.

Note: Reals in CA-Realizer conform to the IEEE floating point standard, which means that programs written in CA-Realizer can take full advantage of the speed and precision enhancements offered by math coprocessors.

String Data Types

A string (or *alpha*) is a text value that consists of zero or more of the following: letters of the alphabet, digits, punctuation marks, and other special characters. Strings and string array elements can be up to 65,500 characters long.

CA-Realizer supports two string data types. The first one, **STRING**, is the default data type for strings.

Type	Description
STRING	A variable-length string Variable length strings are available in all modern forms of BASIC, and can contain from 0 to 65,500 characters.
STRING * <i>n</i>	A fixed-length string where <i>n</i> represents the fixed number of characters in the string Fixed-length strings always have the specified number of characters. If the value assigned to a fixed string variable has too many characters, the extra characters are truncated. If the value assigned has too few characters, spaces are added to the end as padding.

String constants must be enclosed within double quotes. Examples of valid string constants are displayed below:

```
"Wherever you go, there you are"  
"2.7182818284"
```

To also include a quote in a string constant, use two pairs of double quotes. For example:

```
"This "word" is a quote."
```

If you assign this string to a variable and print it, you will see
This "word" is a quote.

Date-Time Data Type

The date-time data type in CA-Realizer is slightly different than the other simple data types in that these variables are created and accessed through date-time functions. As their name suggests, they represent a date and a time, for example, November 11, 1968, 1:35 PM. To assign a value to a date-time variable you must use the StrToDate conversion function, as shown below.

```
dStart = StrToDate("01-Jan-94 9:45:32")
```

Or you can use a special shorthand notation for the StrToDate function as shown below.

```
dStart = &D("01-Jan-94 9:45:32")
```

Date-time values may be passed to the StrToDate function using a variety of formats, and must always be enclosed in a pair of double quotation marks, as shown below.

```
"01/01/94 9:45AM"      "01-Jan-94 0945"      "940101"  
"9:45:32"
```

You can also assign the current system date and time to a variable by using the QDate function. Using date-time variables with the related functions makes the manipulation of time-related data much simpler than with other versions of BASIC. It also makes CA-Realizer programs more compatible with data in standard business applications, such as spreadsheets.

Note: Date-time values can range between January 1, 1900, and December 31, 2099.

Complex Data Types

Arrays

An *array* is a group of data of a single specific data type representing a set of data values. CA-Realizer supports both fixed and dynamic arrays. An array can have one, two, or more dimensions. In a fixed array the number of dimensions is specified in the DIM statement, and in a dynamic array, the number is defined by the first assignment statement.

Data Type	Description
ARRAY (<i>Dims</i>) AS <i>Type</i>	A fixed array of elements. The number of dimensions, data type of the elements, and number of elements can not be changed dynamically
ARRAY	A dynamic array of elements. The number of elements can be changed at any time. The number of dimensions and data type are fixed once defined.

All the elements of an array must contain values of the same data type. In the case of fixed arrays, the allowable data types include any of the 11 simple data types or 4 complex data types. The data type for a fixed array is specified in a DIM statement. For dynamic arrays, the data type of the elements may be one of the 4 default simple data types (LONG, DOUBLE, STRING, or DATETIME) or may be one of three complex data types (dynamic array, record, or family). The data type and number of dimensions for a dynamic array are defined the first time you assign a value to an element.

Records

Whereas all the elements in an array must be of the same data type, a record is a fixed collection of one or more different data types. The components of a record are called fields. A record structure is defined with a TYPE statement, which allows you to specify the data type of each of the fields. You can create a record variable by using the DIM statement and the name of a record structure.

A record variable and its fields form a logical grouping of related data. One accesses the data by specifying the name of the record and the name of the field, separated by a period (for example, *EmployeeName.Address* or *EmployeeName.StartDate*).

Records are often used as the elements of an array, to allow easy access to similar information for many entities. For example, an array of records could be used to store information about all the employees of a company.

For records, the data type of a field may be one of the 11 simple data types or a complex data type (fixed array, dynamic array, another record structure, or a family). Record values can also appear in multi-level data structures involving several complex data types.

Families

Although a family is similar to a record in that its components may include different data types, its structure is not fixed. While a record structure is fixed by the TYPE statement, the components of a family are defined dynamically, or "on the fly". The components of a family are called members. The data type of each family member is defined by the assignment statement that assigns a value to it. Thus, a family is similar to a dynamic array in that the number of components can be increased; however it is different in that the data type of a member can be changed.

A family and its members form a logical grouping of related data, similar to a record and its fields. If the family has one or more members, you can access a data component by specifying the name of the family and the name of the member, separated by a period (for example, *CompanyName.Address* or *CompanyName.Size*). It is possible to have a family with no members.

For families, the data type of a member may be one of the 4 default simple data types (LONG, DOUBLE, STRING, DATETIME) or one of three complex data types (dynamic array, record, or another family). Families can also appear in multi-level data structures involving several complex data types.

Nested Complex Data Structures

CA-Realizer permits you to create complex multi-level data structures in which the data components of a complex data type are simple or complex data types. Instead of defining arrays of integers or reals, you can define arrays of records. And one of the fields in the record could be a family, with the data values of interest being kept as members of this family.

The data types allowed as components for each type of complex variable are listed below.

Data Type	Description
Fixed Array	Simple data types (11), fixed arrays, dynamic arrays, records, families.
Dynamic Array	Default simple data types (4), dynamic arrays, records, families.
Record	Simple data types (11), fixed arrays, dynamic arrays, records, families.
Family	Default simple data types (4), dynamic arrays, records, families.

Note that dynamic arrays and families may not include fixed arrays or simple data types other than the 4 default data types. Fixed arrays and records may include any of the 11 simple data types or 4 complex data types as components.

With a multi-level data structure involving complex data types, you can access the desired data values by specifying the appropriate index numbers for the array, the field name for the record, and the member name for the family. CA-Realizer provides a straightforward syntax for accessing such data. For example,

```
NumValue = Arrayname[4, 7].Fieldname.Membername
```

This statement implies that an array contains a record as an element, and the record contains a field that is a family. The member of the family is assigned to the *NumValue* variable.

Variables and Constants

Variables are used to store data values in your computer's memory during the execution of a program. The data contents of a variable may be accessed by using the name of the variable, and also may be changed by a program. For instance, names from a database or numbers from a spreadsheet may be stored in memory variables and modified. Constants are values that remain fixed during the execution of a program.

Variables

In the CA-Realizer environment, variables are used to store data. A simple variable, sometimes called a scalar, can hold a single data value. An array variable can hold a set of values—such as a spreadsheet column or an entire table.

Note: The set of variables created by a program is maintained in your computer's memory by CA-Realizer even when the program stops running, as long as the CA-Realizer application is not closed.

A variable can contain existing information imported from other applications or new information that is typed in or calculated. Sometimes a value in a variable is saved only temporarily, as a control flag for an operation or as the intermediate result of a calculation.

Unlike many languages that require variables to be explicitly declared before they can be used, CA-Realizer creates a variable the first time it occurs in an assignment operation. Because CA-Realizer BASIC is a dynamic language, variables in general do not need to be declared—rather, their type is inferred when values are assigned to them. The data type may even be changed by a subsequent assignment statement.

Valid Variable Names

Variables are used to store data, which may be accessed through the variable's name.

Legal variable names can contain any of the following characters: A–Z, a–z, 0–9, %, _, @, \$, !, and ?. The names must also meet the following requirements:

- They cannot contain spaces or start with a number.
- They cannot exceed a maximum length of 240 characters.
- They cannot be the same name as any of the CA-Realizer reserved words (for a list of CA-Realizer reserved words, see Appendix A and Appendix B of this guide).

Note that variable names in CA-Realizer are not case-sensitive. For example, the variable named *ProductNum* is the same as *PRODUCTNUM* and *pRoDuCtNuM*.

Some valid variable names are:

```
A
JakeElwood2
!Yo!
```

Examples of illegal variable names are:

A+	Illegal character used
7Down	Begins with a number
Thin Duke	Contains a space
PictureNew	CA-Realizer keyword

Naming variables to reflect their contents makes program logic easier to follow and prevents confusion when you are manipulating them. Therefore, it is recommended that you choose variable names carefully.

Assigning a Value to a Variable

To assign a value to a variable, use the assignment operator, which is the equal sign (=). The general format is as follows,

```
VariableName = Value
```

where *VariableName* is the name of the variable to be assigned the value represented by *Value*.

For example:

```
caption = "Hello"
copies = 5
copies = 5.5
```

Note that the numeric value assigned to a variable may be changed from integer to floating point as needed, and the implied data type is automatically changed. In addition, strings are allocated implicitly and dynamic arrays expand automatically.

All variables remain undefined until they are assigned values. CA-Realizer reports an error if you attempt to use an undefined variable.

Variable Declarations

When using an *implicit* variable declaration, the *data type* of a variable is determined by the value assigned to it. For example, the following assignments,

```
caption = "Hello"  
copies = 5  
price = 9.99
```

create a variable named *caption* which is automatically considered to be a string, a numeric integer variable (LONG) named *copies*, and a numeric real variable (DOUBLE) named *price*. With an implicit variable declaration, one of the four default data types is used

Note: Any value that is a valid CA-Realizer data type can be assigned to a variable—for example, you can create numeric, string, date-time, array, or family variables.

With an *explicit* variable declaration, the type of the variable is determined by the data type assigned to it. Explicit typing of variables requires you to declare the variable with the DIM statement *before* using the variable. For example, you would have to use the following lines

```
DIM caption AS STRING  
DIM copies AS INTEGER
```

before you could assign values to the *caption* and *copies* variables.

Changing a Variable's Type

The data type of an implicitly-declared variable can be changed "on-the-fly" by simply reassigning it a different value. Note that you can dynamically change the data type of implicitly-declared variables only, not the data type of explicitly declared variables. Explicitly-declared variables require that you first use the CLEAR statement. (See the Clearing Variables section in this chapter and the *CA-Realizer Language Reference Guide* for more information about the CLEAR command.)

Valid assignment statements that change a variable's type are presented below. In the case of arrays, the data type of elements can not be changed, with one exception - changing a LONG integer array to one that contains DOUBLE real numbers.

```

Foo = 5                'Foo is a numeric.
Bar = Foo              'Bar is a numeric.
Foo = "Hello"          'Foo is now a string, that
                        'replaces the previous value.
Junk[6] = 9            'Junk is an array of LONG
                        'integers.
Junk[2] = 7.5          'Junk is now an array of
                        'DOUBLES.
Foo[2] = 7             'Foo is now an array of reals.
                        'Previous value/type is lost.
Junk = 9               'Junk is now a numeric scalar.
                        'Entire array is lost.
```

If two arrays were declared as follows,

```

Blah[1] = 9
Abc[1] = "Blah"
```

the following would be illegal assignment statements:

```

Blah[1] = "Gleep"      'You cannot change the type
                        'of an array.
Abc[2] = 7              'All array elements must be
                        'of the same type.
```

Obtaining Information About a Variable

The QVar function returns information about a variable, such as its type, whether it is defined, whether it is an array or a scalar, or how many dimensions an array has. The general form is:

```
ns = QVar(Data, Query)
```

For example:

```
IF QVar(Balance, _Defined) THEN
    NewBalance = Balance * (1 + InterestRate)
ELSE
    NewBalance = 0
END IF
```

This example confirms that the *Balance* variable is defined before it is used in an expression. Referring to undefined variables causes an error.

The type of a variable can be checked with the QVar function as illustrated below:

```
IF QVar(Price, _Alpha) THEN
    Total = Total + StrToNum(Price)
ELSEIF QVar(Price, _Real) THEN
    Total = Total + Price
END IF
```

This example checks to see if *Price* is a string and, if it is, converts it to a real with the StrToNum function.

The StartValid and EndValid functions find the valid range of an array. They return the number of the first valid element and the last valid element, respectively.

```
ns = StartValid(Data [, DimNum])
ns = EndValid(Data [, DimNum])
```

If *Data* is a scalar or an undefined value, the result is 0. For multi-dimensional arrays, use *DimNum* to indicate the dimension. (If *DimNum* is not specified, the default is 1). To determine the number of dimensions, use

```
nd = QVar(StartValid, _NumDims)
```

Clearing Variables

Variables, arrays, procedures, and functions all use memory. An 8000-element array, for example, takes up 64K while it is defined. Erase variables using the CLEAR command in order to free this memory.

For example:

```
PROC MyProc
.
.
.
DIM copies AS INTEGER
.
.
.
CLEAR MyProc, copies
```

CLEAR is supplied a list of variables, procedures, and functions to clear. You can mix variable, procedure, and function names on a single line. Both implicitly and explicitly declared variables can be cleared with the CLEAR command. You can also clear a record variable, a family, or a member of a family. If a family is cleared, all members of that family are cleared as well. User-defined types and record structures cannot be cleared.

Exchanging Values

Use the SWAP command to exchange the values of two variables. If either variable has been defined with an explicit type, the value being assigned to it must have the correct type and range.

For example,

```
SWAP a, b      'Assign the value of b to a and
               'a to b
```

is functionally equivalent to:

```
tmp = a
a = b
b = tmp
```

Constants

CA-Realizer supports both *symbolic* and *predefined* constants.

Symbolic Constants

A symbolic constant is a name you use to represent an unchanging data value. You can create symbolic constants by prefixing a typical assignment statement with the `CONST` keyword as follows:

```
CONST VariableName = Value
```

For example:

```
CONST PI = 3.14159
```

Constants implicitly use one of the four default simple data types (`LONG`, `DOUBLE`, `STRING`, `DATETIME`) depending on the nature of the assignment statement. However, once you have defined a constant, it is illegal to try to assign a new value to it, or to use the name as an array, record, or family name, unless you `CLEAR` it first.

Predefined Constants

You can use predefined constants in your programs as parameters and modifiers in many of the CA-Realizer commands and functions.

These CA-Realizer commands have parameters that tailor the way the command operates. For example, `FileOpen` has a parameter indicating whether the file to be opened can be written to, and `ChartSetColor` has a parameter that selects the line color. These parameters are expected to be numeric values.

To increase the readability of these commands, there are predefined constants that give names to these values. All predefined constants start with an underscore (for example, `_Write`, `_Blue`, and `_DI_Month`).

For a complete list of the CA-Realizer predefined constants, see Appendix B in this guide.

Data Typing in CA-Realizer

When creating variables, CA-Realizer supports both *implicit* and *explicit* data typing. Both are explained in greater detail below.

You can create your own data types, using the TYPE statement, so you can give new names to predefined data types to help make programs easier to read. You can also use the TYPE statement to define record data types. Instructions for accomplishing these tasks are presented in this section.

Implicit vs. Explicit Data Typing

Traditionally, interpretive languages like BASIC allow fully dynamic, undeclared, untyped, and unscoped variables. This is known as *implicit* data typing: when assigning a value to a variable for the first time, you are not required to declare it—instead, its type is *inferred* when a value is assigned to it.

For example, a programmer can write

```
a = 5
```

without worrying about declaring *a* or defining its type (CHAR, INTEGER, SINGLE, etc.). This is very convenient—it allows developers to see results quickly without worrying about technical details. Informal, dynamic data handling is one of the chief advantages of the BASIC language.

In keeping with the BASIC tradition, CA-Realizer permits implicit data typing of variables, allowing you to ignore most of the technical considerations associated with internal data handling.

Note: See Assigning a Value to a Variable in this chapter for more detailed information about implicitly assigning a variable's type.

However, there are times when you may want to *explicitly* declare the type of a value. For example, explicit data typing is extremely helpful when tuning programs for optimal speed and efficiency.

You can create a variable of a specific type by declaring it with the DIM statement as discussed below.

The DIM Statement

To explicitly declare a CA-Realizer variable, whether it is a scalar or an array, use the DIM statement as follows:

```
DIM [LOCAL] Var1 [(Subscripts)] [AS Type]
    [, Var2 [(Subscripts)] [AS Type] ...]
```

If the LOCAL keyword is used, the variables that are created are local to the current procedure, function, or program file.

Var1 is the name of the variable to be declared and *Type* is a keyword representing the CA-Realizer data type for that variable.

For example:

```
'Declares x to be an integer
DIM x AS INTEGER
```

```
'Declares y as a 20 character fixed size string
DIM y AS STRING * 20
```

```
'Declares z to be a fixed array of 5 doubles
DIM z (1 TO 5) AS DOUBLE
```

Note the use of the data type keywords presented in the CA-Realizer Data types section earlier in this chapter.

Once a variable has been declared with a specific type, it is illegal to try to change its type without first clearing the variable with the CLEAR statement.

Also, CA-Realizer reports an error if the value of a declared variable exceeds the valid range of its assigned data type. For example:

```
DIM x AS INTEGER, y AS DOUBLE
x = 40000      'Causes an error: x cannot exceed
               '32767
y = "Foo"      'Causes an error: cannot change
               'type of y from DOUBLE to STRING
```

Note: To create an initialized fixed size array, the array's dimensions (called *Subscripts*) should be specified as shown in the DIM syntax presented at the beginning of this section. Up to 30 dimensions can be specified, separated by commas. The default lower bound of 1 can be changed to 0 with the OPTION BASE command. Arrays are discussed in detail later in this chapter.

For more information about the DIM command, see the *CA-Realizer Language Reference Guide*.

Creating User-defined Data Types

In addition to explicitly declaring variables with the standard CA-Realizer data types, you can also create your own data type names for use in variable declaration statements. Using the TYPE statement, you can give new names to the predefined data types to help make programs easier to read. In this way you can use the new name in the DIM statements for the variables, and if you should want to modify the data type in the future, you need only change the single TYPE statement.

The general form of the TYPE statement is:

```
TYPE NewType AS Type
```

where *NewType* is the new name to be used for the predefined data type given in the *Type* argument. (All CA-Realizer data types are valid in the *TYPE statement*.)

For example, the statement:

```
TYPE ID AS INTEGER
```

declares a new user-defined type named *ID*. *ID* is not a variable, but rather a new, customized name (i.e. an alias) for the `INTEGER` data type that indicates the data type's purpose, as well as the amount of memory used for storage.

After defining this new data type name, you can use it anywhere you would use the data type `INTEGER`. User-defined data types, therefore, simply provide another way of expressing a standard data type.

In addition, once you declare a user-defined type, you can create variables of that type the same way as before. For example, if you have defined the following,

```
TYPE ID AS INTEGER
```

the following statements are identical:

```
DIM empid AS ID
```

```
DIM empid AS INTEGER
```

One advantage to using user-defined data types is that, if at some point the application requires ID numbers to have a larger range of values, you can change the one `TYPE` statement used to define the user-defined data type, rather than having to change declaration statement in the program that refers to the `ID` data type.

Creating and Using Arrays

An array contains elements of a single data type, organized according to one or more dimensions, specifically:

- As a vector (a one-dimensional array)
- As a table or matrix (a two-dimensional array)
- As a higher-dimension array (such as sales data stored as a four-dimensional array indexed by product, month, year, and salesman)

Each value in an array is called an *element*, and each element is addressed by one or more *index numbers*, also called *indices*. There is an index number for each dimension. The number of elements in an array is determined by taking the product of the number of possible values for each index number.

Multi-dimensional arrays in CA-Realizer are always uniform in shape, and are not “ragged”. This means that for any row or column in a two-dimensional array, the number of elements in each row or column is the same. The corresponding table is a full rectangle. For example, in a 10 x 20 two-dimensional arrays, each of the 10 rows has 20 elements. It can also be said that each of the 20 columns has 10 elements. There are 200 elements in the array.

With more than two dimensions, a similar property also holds. If you fix the index numbers for one or more dimensions while letting the index numbers for the other dimensions range over all valid values, the number of elements in such a subrange is always the same. In a three-dimensional array, the corresponding array of cells is a full “cube” with no ragged edges.

By default, the first element of a one-dimensional array has an index of 1, the second an index of 2, and so on. If the array has more than one dimension, the first index represents the outermost dimension (for example, the row), and the second index represents the second dimension. If there are N values for the second index, the elements of the array are numbered as (1, 1), (1, 2), . . . (1, N), (2, 1), (2, 2), . . . , (2, N), (3, 1), (3, 2), . . . and so on.

CA-Realizer arrays can support up to 30 dimensions, and can have either a *fixed* (static) or *variable* (dynamic) size. In either case, the number of dimensions of an array is fixed. The number of dimensions and range of index values for a fixed array is defined by a DIM statement. The number of dimensions for a dynamic array is defined the first time a value is assigned to an element by the specified number of indices for the element.

The General Form of an Array

The general form of an array reference is:

```
ArrayName[Index1, Index2, ..., Index30]
```

Index numbers are separated by commas and enclosed in brackets. However, to provide flexibility, you can optionally substitute parentheses for the brackets. For example:

```
Price[20]  
Sales(10, 20)  
PtColor[10, 20, 30]
```

If only one index number is provided, the array is a *vector*. If there are two index numbers, the array is a *table* or *matrix*. When the elements of an array are to be stored in rows and columns, the array must be two-dimensional.

The following example creates three arrays—*Name*, *Price*, and *Quantity*—that use a set of imaginary data from the sales report of an imaginary company:

```
Name[1] = "rock"  
Name[2] = "round stone"  
Name[3] = "square stone"  
Name[4] = "metal scissors"  
Name[5] = "heavy paper"  
  
Price[1] = 2.30  
Price[2] = 3.75  
Price[3] = 10.50  
Price[4] = 1.25  
Price[5] = 7.75  
  
Quantity[1] = 25  
Quantity[2] = 33  
Quantity[3] = 6  
Quantity[4] = 10  
Quantity[5] = 7
```

The *Name* array is a string array, *Price* is a real array, and *Quantity* is an integer array. Each array has five elements, each of which is accessed by the index number within brackets.

The array indices in this example are numbered starting with 1. Thus, *Price*[1] is the first element in array *Price*, *Price*[2] is the second element, and so on. Index numbers for arrays can begin with any non-negative integer value as the first element (1 is the default starting value, which can be changed to 0 if so desired with the `OPTION BASE` command). For example, CA-Realizer arrays can have data values for elements with index numbers between 10 and 50.

The range of index values for each of the dimensions is called the array's *valid range*. You can use the Variable Display option in the Debugger to view the valid range. For example, enter the following line into a program window, highlight it, and select **INSTANT** from the **RUN** menu:

```
MyArray[5:10] = "foo"
```

This array will have 6 elements, with a valid range between 5 and 10. Now bring up the Variable Display by selecting **SHOW VARIABLES** from the **RUN** menu. You can see that the array begins with element 5.

For more information about the Variable Display, see the *CA-Realizer User Guide*.

Constructing and Concatenating Arrays

As mentioned previously you can assign a data value to a single element of an array by means of a simple assignment statement. This is somewhat tedious if you want to assign many data values at the same time. To make this easier, CA-Realizer provides a simple method for assigning several values to an array or to a subrange of an array using a single subrange statement.

The general form for assigning several values to an array using the *array constructor* is:

```
ArrayName = {Value1, Value2, ... ValueN}
```

The elements of the array are listed between the { } braces and are separated by commas. Each item in the list pertains to a single element in the array—the first item is the first element, the second item is the second element, and so on. An array constructor list can contain variables and expressions as well as constants and arrays.

For example, the following code defines the same three arrays as in the example in the previous section, using only 3 statements:

```
Name = {"rock", "round stone", "square stone",  
        \ "metal scissors", "heavy paper"}  
Price = {2.30, 3.75, 10.50, 1.25, 7.75}  
Quantity = {25, 33, 6, 10, 7}
```

Concatenating Arrays

You can also use the array constructor to concatenate arrays and elements. For example you can add or append elements to an array or append one array to another. For example, given the following:

```
PartA = {1, 3, 7}  
PartB = {9, 11}  
PartC = {PartA, PartB, 13}
```

PartC is a 6-element array that is created by concatenating two other arrays and an additional element. It contains the values: 1, 3, 7, 9, 11, and 13.

Special Array-handling Features Unique to CA-Realizer

CA-Realizer provides many array-handling enhancements over earlier versions of BASIC. You should be aware of these enhancements, particularly if you are converting existing code to CA-Realizer.

Enhancements have been made in the following array-related areas; they are described in detail in the following sections:

- Subranging
- Dynamic expansion
- Assignments between arrays with different numbers of dimensions

Array Subranging

Typically, earlier BASIC implementations populated arrays using one of the following three methods:

- DATA statements
- Explicitly assigning a value to an array element
- Using loops

For example:

```
FOR i = 97 to 122
    MyArray(1, i) = 1
NEXT i
```

CA-Realizer does not use DATA statements; instead it provides the array constructor. For example:

```
A = {"Craig", "Mike", "Ed", "Ivan"}
```

Furthermore, CA-Realizer has added a subrange operator (:) to simplify the manipulation of the subranges (or subsections) of an array. For example, the following syntax produces the same result as the loop presented above:

```
MyArray(1, 97:122) = 1
```

Subranging offers a very powerful and efficient way to work with arrays. In addition to populating arrays with data, you can also use subranging to reference parts of arrays, selectively insert and delete elements, and even transpose matrices.

Try the following example:

```
A = Index(10)^2
ROWPRINT A[3:7]
```

In this example, the Index function generates an array of 10 elements ranging from 1 through 10, which are then squared. Subranging allows you to easily extract 4 elements from this array. In general you can reference a subrange of elements from any part of an array.

You can also use the array constructor to assign elements to a subrange of index values in an array. The following statement assigns values for index values ranging from 5 to 8 for the first dimension, and index values ranging from 2 to 3 for the second dimension.

This subrange of the array's valid range has 8 elements, since the range for the first dimension has 4 numbers and the second has 2. The array constructor must have the same number of elements as required to match the subrange. The first and third statements are both valid since the array constructor contains 8 elements in either case.

```
ArrayA[5:8, 2:3] = {1, 2, 3, 4, 5, 6, 7, 8}
PartA = {1, 3, 7}
ArrayB[5:8, 2:3] = {PartA, 4, 5, 6, 7, 8}
```

Removing Elements

Subrange can also be used to reduce the number of elements in a dynamic array, as shown below.

```
A = {5, 3, 2, 1, 9, 6, 7}
ROWPRINT A
5 3 2 1 9 6 7

A[4:6] = A[5:7]
A = A[1:6]
ROWPRINT A
5 3 2 9 6 7
```

Here, the fourth element, the value 1, is removed by copying elements 5 through 7 over elements 4 through 6. The array is then reduced by one element by dynamically redimensioning it for 6 elements in the fourth line of code (as explained in Dynamic Array Expansion section presented later in this chapter).

Inserting Elements

Note that the removal process can be easily reversed, as demonstrated below:

```
B = {5, 3, 2, 9, 6, 7}
ROWPRINT B

5 3 2 9 6 7

B[5:7] = B[4:6]
B[4] = 1
ROWPRINT B

5 3 2 1 9 6 7
```

Transposing Elements

The following multi-dimensional example is often encountered in scientific and engineering programming:

```
A[1, 1:3] = {1, 2, 3}
A[2, 1:3] = {4, 5, 6}
A[3, 1:3] = {7, 8, 9}
PRINT A

1 2 3
4 5 6
7 8 9

FOR i = 1 to 3
    B[i, 1:3] = A[1:3, i]
NEXT i
PRINT B

1 4 7
2 5 8
3 6 9
```

To transpose the matrix, you first assign data values with the array constructor, then use a loop to swap the rows with the columns. This produces the same result as using the CA-Realizer MatTranspose function as follows:

```
PRINT MatTranspose(A)
1.0000      4.0000      7.0000
2.0000      5.0000      8.0000
3.0000      6.0000      9.0000
```


Dynamic Array Expansion

Earlier versions of BASIC specifically required you to declare an array using the DIM statement. For example:

```
DIM SomeNumbers(12)
```

Once declared, the range of index numbers for a fixed array could only be changed by explicitly redimensioning the array using the REDIM command, in which case all of the data in the array was lost. This problem can be avoided by using dynamic arrays.

In CA-Realizer, a DIM statement is not needed when you work with dynamic arrays. The only time the DIM command is required is to declare a static array of elements that have a fixed type.

While the number of dimensions of a dynamic array cannot be changed, the range for an index can be adjusted dynamically as required. The following statements populate the *SomeNumbers* array with 5 random numbers (the Rnd function is used to generate the random numbers):

```
SomeNumbers[5:9] = Rnd(5)  
PRINT SomeNumbers
```

```
0.7515  
0.3456  
0.1690  
0.6573  
0.4919
```

If you now assign an element with an index number beyond this range (for example, 12 since the current valid range is 5 through 9), you can see that CA-Realizer extends the range to include the new element:

```
SomeNumbers[12] = Rnd
PRINT SomeNumbers
```

```
0.7515
0.3456
0.1690
0.6573
0.4919
0.0000
0.0000
0.0635
```

Since a non-existent element (with index number 12) was assigned a value, CA-Realizer adjusted the valid range of the array from 5 to 9 to 5 to 12.

In addition, CA-Realizer also permits you to immediately assign and access the value of a new element. In the above example, the number generated by the Rnd function is assigned to element 12 and is printed in the output (0.0635).

Note that the undefined elements (10 and 11) are initially assigned values of 0. By default, CA-Realizer assigns all new undefined elements (those between the existing range and the new element) one of the following data values: 0 for integers or reals; an empty string ("") for strings; and 1/1/1900 00:00:00 for date-time values).

You cannot access elements 1 through 4 because they are outside the valid range. For example, the following statement is illegal:

```
PRINT SomeNumbers[2]
```

Try the following example—it creates a two-dimensional array with 6 elements:

```
A[1:2, 1:3] = 4
PRINT A
```

```
4  4  4
4  4  4
```

The next statement expands *A* to a 4x4 two-dimensional array with 16 elements:

```
A[4, 4] = 16
PRINT A
```

```
4  4  4  0
4  4  4  0
0  0  0  0
0  0  0  16
```

Notice how the unassigned new elements in the expanded array are filled with zeros.

Note: The only time the DIM command is required, is to declare a static array of elements that have a fixed type. For example:

```
'Declare z to be a fixed array of doubles
DIM z(1 TO 5) AS DOUBLE
```

Assignments Between Arrays with Different Subranges or Dimensions

CA-Realizer allows you to assign values between different subranges of arrays, and even between arrays with different dimensions. You can use subrange notation to concisely define which elements of an array are to be involved in the assignment. This capability provides exceptional power and convenience.

For instance, you can write a statement to generate a two-dimensional array from a one-dimensional array as follows:

```
A [1:9] = Index(9)
B[1:3, 1:3] = A [1:9]
PRINT B
```

1	2	3
4	5	6
7	8	9

Or you can even use a 3-element subrange of array *B* in an expression defining a 5-element subrange of array *A*.

```
A[2, 1:5] = {B[3, 1:3], 7, 9}
```

CA-Realizer requires that the number of elements on the right side of an assignment statement equal the number of elements on the left. You must multiply the number of choices for each dimension in a subrange to determine the number of elements.

Here is an example where an entire 4 x 3 two-dimensional array A is assigned to the elements in a 6 x 2 array B.

```
A[1:4, 1:3] = Index(12)
```

```
PRINT A
```

```
1  2  3
4  5  6
7  8  9
10 11 12
```

```
B[1:6, 1:2] = A
```

```
PRINT B
```

```
1  2
3  4
5  6
7  8
9  10
11 12
```

This next statement assigns the 12 elements of the two-dimensional array A to a subrange of a three-dimensional array C. In this assignment statement there are 12 elements on both sides of the equals sign.

```
C[1:2, 6:7, 3:5] = A
```

Operating on Arrays

CA-Realizer operators (e.g., mathematical, string, date-time) and functions that operate on a single array accept arrays with any number of dimensions. The result is usually an array with the same number of dimensions and bounds.

When operating on more than one array, the number of dimensions for each array must be the same, and—as with one-dimensional arrays—the operation applies to the intersection of the valid ranges.

For example, try the following sample program:

A is a 3 x 5 array with a valid range of (1:3, 1:5) and B is a 6 x 4 array with a valid range of (2:7, 3:6). The final assignment statement creates a new array C of two dimensions which has an index range that is common to both arrays (that is, the range of index values is the intersection of the other index ranges).

The common index range for the first dimension is from 2 to 3, and for the second dimension it is from 3 to 5. Thus C will be a 2 x 3 two-dimensional array that is valid for the index range (2:3, 3:5).

```
A(1:3, 1:5) = Index(15)   'a 3 x 5 array
B(2:7, 3:6) = Index(24)   'a 6 x 4 array
C = A + B                  'a 2 x 3 array
```

PRINT A

```
1      23      4      5
6      7      8      9     10
11     12     13     14     15
```

PRINT B

```
1      2      3      4
5      6      7      8
9     10     11     12
13     14     15     16
17     18     19     20
21     22     23     24
```

PRINT C

```
9     11     13
18     20     22
```

With one-dimensional arrays, if you use the `Index` function to define an index range, and then reassign a subrange of the array to be the entire array, the result is a similar array representing only a portion of the original array of 6 elements:

```
D = Index(6)
D = D[2:4]
ROWPRINT D
2 3 4
```

Likewise, with multi-dimensional arrays, subranges can be extracted by specifying a range for each index number:

```
X = B[2:3, 4:5]
PRINT X
2 3
6 7
```

If any of the indices are not ranges but are instead a single index number, the resulting array has fewer dimensions than the original. For example, if `C` is a 4 × 4 array, the following statement

```
E = C[3, 1:4]
```

produces a one-dimensional array `E` with all 4 of the elements in the third row of `C`.

In general, the number of dimensions decreases by one for each index that is specified as a single index number instead of a range. For example, `F` is a one-dimension array, while `C` has two dimensions:

```
C[1:4, 1:4] = Index(16)
F = C[3, 1:4]
ROWPRINT F
9 10 11 12
```

Arrays of Arrays (Subarrays)

CA-Realizer also allows you to create an array of arrays. In this case, each element of a fixed array may be either a fixed or dynamic array. Each element of a dynamic array will be another dynamic array. Each of the dynamic subarrays (or nested arrays) of the array can have a different, dynamically changing number of elements, and each subarray can even contain values of a different data type.

As with all arrays, an array of arrays can be created by simply assigning the name of an array as a single element of a new dynamic array. In this case, however, the value being assigned is itself an array.

```
A = {5, 8, 3, 2}
Z[1] = A
```

These two lines create two dynamic arrays. *A* is a one-dimensional, 4-element array of numeric values. *Z* is a one-dimensional array of arrays, whose only element is a one-dimensional, 4-element array of numerics. The elements of an array of arrays are sometimes referred to as *subarrays*.

The elements of a subarray are referenced in the same way as any other array element. The notation, however, may look strange at first (unless you have used the C programming language).

For example, the following example uses the array of arrays named *Z* created in the last example:

```
Z           'The entire array of arrays
Z[1]        'The element of Z, which is
            'the subarray {5, 8, 3, 2}

Z[1][1]     'The first element of the subarray,
            'which contains the value 5

Z[1][2]     'The second element of the subarray,
            'which contains the value 8
```


Using this notation, elements of the subarrays can be referenced in the same way as any other array, including the use of subranges. The subarrays can also be expanded dynamically. For example,

```
Z[1][5] = 22
```

expands the only element in array *Z* from a one-dimensional 4-element array to a one-dimensional 5-element array.

New subarrays can be added to the array by assignment. The array of arrays will expand as expected. For example:

```
Z[2] = {"Run", "Kick", "Pass"}
```

Z is now a 2-element array of arrays. The new element is a string array with three elements.

As you can see, each subarray in an array of arrays can have a different type. In fact, each element can have a different number of dimensions. The subarray can even be an array of arrays. For example, the following statement,

```
Z[3] = Z
```

adds another element to *Z*, which is an array of arrays containing the two subarrays in *Z*. *Z* now contains the following arrays:

```
Z[1]           {5, 8, 3, 2}
Z[2]           {"Run", "Kick", "Pass"}
Z[3][1]        {5, 8, 3, 2}
Z[3][2]        {"Run", "Kick", "Pass"}
```

You can reference single data values accessed via the third element of *Z* by also specifying the index number for the first subarray as well as the index number for the element in the lowest level subarray, as follows:

```
Z[3][1][4]     2
Z[3][2][2]     "Kick"
```

In general, subarrays can be treated in the same way as any array variable. You must specify a set of index numbers to access one of its elements, or a subrange of index numbers to specify more than one element. With CA-Realizer's dynamic array expansion and automatic array processing, extremely complex and powerful data structures can be created. And these structures can be described with great efficiency and simplicity by using subrange notation.

Limits on Sizes of Arrays

Array index numbers can range from 0 to 2,147,483,647, but the total size of the array in your computer's memory can be no larger than approximately 65,500 bytes. The number of elements allowed in an array depends on the type of the data that the array is storing, regardless of the number of dimensions.

With dynamic arrays, numeric values are stored as long integers (4 bytes each) or double-precision floating point values (8 bytes each), depending on whether any large or fractional values are assigned to the array. If an array has been declared as a fixed-size array, the number of elements as well as the data type never change (for more information, see the DIM command in the *CA-Realizer Language Reference Guide*).

The following table presents the maximum number of array elements and element size per element type:

Element Type	Element Size (in bytes)	Maximum Number of Array Elements
BYTE, CHAR	1	65,500
WORD, INTEGER	2	32,735
DWORD, LONG	4	16,368
SINGLE	4	16,368
DOUBLE	8	8,184
STRING	4	16,368
STRING * <i>n</i>	<i>n</i>	65500 / <i>n</i>
DATETIME	8	8,184
ARRAY	4	16,368
RECORD (<i>m</i> fields)	<i>m</i>	65500 / <i>m</i>
FAMILY	4	16,368

When dynamic strings, arrays, families, or records are used as array elements, the actual array in memory contains only a pointer to the actual data (the elements). This allows a larger number of elements in the array, regardless of the length of a string, the number of elements in a subarray, the number of fields in a record, or the number of members in a family. Each element in an array of strings can be up to 65,500 characters long.

As noted above, an array of arrays can contain up to 16,368 elements, each of which is an independent array that can contain its own maximum number of elements. With arrays of arrays, it is possible to create multi-dimensional arrays that have essentially no limit on the total number of elements.

For example, if an array of arrays contained 16,000 subarrays, and each of these had 16,000 integer elements, this would represent a two-dimensional array with 256 million elements, requiring half a gigabyte of memory to store. If the memory is available, CA-Realizer can create and access such an array.

Printing Arrays

When printing arrays in CA-Realizer, keep the following simple and useful rules in mind:

- To PRINT an entire array, specify the array without the bracketed array index values or ranges. This displays the values of each array in separate columns. (When index values or ranges are omitted, most CA-Realizer commands act upon the entire array.)

- When more than one array is printed, elements of the same range are printed on the same row.

- Print a subrange of an array by using the subrange syntax:

```
PRINT Price[2:4]
```

- It is also possible to mix scalars and arrays in a PRINT command. A scalar is repeated on every printed row generated by this command:

```
PRINT "Product Name: ", Name
```

For a more detailed discussion of printing capabilities see the Printing section of the Controlling Input and Output chapter.

Creating and Using Records

Suppose you want to write an application that works with groups of related data values; the data values may be of different data types but are all related in some way. To quickly accomplish this, CA-Realizer allows you to create a structure—called a *record*—that conveniently stores an entire collection of related data under a single variable name.

The record data type generalizes upon the concept of a fixed array, whose elements have a single data type. A record structure in CA-Realizer allows you to create a logical grouping of different data types. You create a record structure by using a TYPE statement to define the record structure's name and the name and type of each field. This complex data type allows you to define related data components, which themselves may be one of the 11 simple data types or even one of the 4 complex data types (fixed array, dynamic array, another record structure, or a family). Records can also appear in multi-level data structures involving several complex data types

Records are simply collections of data components with different types. The data types of the fields in a record structure are declared explicitly and may not be changed once defined. You can access a single field of a record, or manipulate entire records as a whole, assign them to one another, and pass them as parameters to procedures and functions.

Defining a Record Structure

Before you can create an instance of a record, you must define the record's structure—that is, you must declare the names and data types of the *fields* that should be part of the record. The term *field* is used to refer to the individual elements in a record.

To define a record's structure, use the TYPE statement as follows:

```
TYPE RecordStructureName
    FieldName1 AS Type
    FieldName2 AS Type
    .
    .
    .
    FieldNameN AS Type
END TYPE
```

where *RecordStructureName* is the name of the new record structure, *FieldName1...N* are the names of the individual fields for the record, and *Type* is any valid CA-Realizer data type (including another record type or family).

The TYPE statement below creates a record structure with 8 fields:

```
TYPE Employee
    Name AS STRING * 40
    Id AS INTEGER
    Address AS STRING * 80
    City AS STRING * 20
    State AS STRING * 2
    Zip AS LONG
    Salary AS SINGLE
    SalaryHistory AS ARRAY
END TYPE
```

Note that the data contents of each of the fields of a record variable are initialized when the variable is declared using a DIM statement. If one of the fields in the record is a dynamic array, its bounds and number of dimensions are undefined until the field is assigned an array name.

Once you create a record structure you can use it to define the type of an array. Note that you must use the name of a record structure in the following statement, rather than simply use the word `RECORD`. The word `RECORD` is not a reserved CA-Realizer keyword like the names of the other data types.

```
Dim SalesDept(1000) AS Employee
```

Declaring a Record Variable

Once a record structure has been defined, you can declare one or more different variables based on that structure by using the `DIM` statement. For example, if you defined the record structure shown in the previous example, you could then legally declare two or more record variables (*Manager* and *Supervisor*) of type *Employee* as follows:

```
DIM Manager AS Employee  
DIM Supervisor AS Employee
```

However, once a record instance is created, you can not modify its parent record structure and have it reflected in either *Manager* or *Supervisor*. And you can not change the name of its parent record structure from *Employee* to a different structure such as *Customer*. The only way to accomplish this would be to create a different structure (*EmployeeX*), then create a different record variable (*ManagerX*) and use assignment statements to load the data components of *Manager* into *ManagerX*.

Once created, a record variable exists until it is eliminated by using the `CLEAR` command. You can not clear a single field of a record variable.

Note: As with all declared variables, the type of a record variable **cannot** be changed once you create it using the `DIM` statement. Thus, you can not subsequently use the name *Manager* to represent a simple scalar variable. The only way to accomplish this would be to first `CLEAR` the record variable *Manager*.

Accessing the Fields of a Record

To refer to a specific field of a record variable, simply use the record name, followed by a period and the field name. For example, the following statements assign values to the various fields—like *Name*, *Address*, and *Salary*—in the *Manager* record:

```
Manager.Name = "Bob Stone"
Manager.Id = 11286
Manager.Address = "123 Powell Drive"
Manager.City = "Vorhees"
Manager.State = "NJ"
Manager.Zip = 80504
Manager.Salary = 45000.00
Manager.SalaryHistory = {43000.00, 40500.00}
```

Arrays of Records

CA-Realizer also permits you to create arrays of records. An array of records can be a very useful structure for sorting and searching operations, and for saving and retrieving data from disk files. They can also simplify the passing of data between functions and procedures.

An array of records is declared using a DIM statement. For example,

```
DIM SalesDept(1000) AS Employee
```

creates a 1000-element array named *SalesDept*. The records of *SalesDept* are created using the *Employee* record structure.

Once you have declared an array of records, you can access the elements of the array using standard subscript notation. For example,

```
FoundIt = SalesDept(34).Name
```

assigns the value of the *Name* field of element 34 of the *SalesDept* array to the *FoundIt* variable.

Records in Nested Complex Data Structures

The previous section showed how records and arrays can be combined to create a two-level nested data structure. Records can also be used to create multi-level data structures involving more than two complex data types. CA-Realizer allows you to create extremely general data structures by using complex data types as the components of other complex data types.

These complex multi-level data structures can be pictured as trees, in which each nested component is represented as a lower level node connected to its (complex) parent. To access data in such a tree structure, you must specify the index numbers for arrays, field names for records, and member names for families that describe the path to the desired (simple) data item.

Extracting a Column from an Array of Records

An array of records can be viewed as a two-dimensional array of varying data types, where the array index is the row number and the field name is the column. Alternatively, the array can be seen as a database table, with the array index as the record number and the field name as the field.

When used for these purposes, it is often useful to retrieve the values for a certain field from each record. The special notation of *ArrayName.Field* extracts these values and returns them as an array. For example:

```
TYPE partID AS INTEGER
    Name AS STRING
    Price as SINGLE
END TYPE

DIM SalesItems(1 to 10) AS partID
.
'Assign values to the fields
.

'Create an array of sales item names
Names = SalesItems.Name
```

Creating and Using Families

Suppose you want your program to work with groups of related data values of different data types, but you know that the number of different data types will be subject to change over time. You need something that is dynamic and flexible, but similar to a record. To accomplish this, CA-Realizer allows you to create a family that conveniently stores an entire collection of related data under a single variable name, with the ability to modify the family structure over time.

The family data type generalizes upon the concept of a record, whose fields are fixed. You create and modify a family's structure "on the fly" by means of assignment statements. This complex data type allows you to define members of the family, each of which may be one of the 4 default simple data types or even one of 3 complex data types (dynamic array, record, or a family). Families can also appear in multi-level data structures involving several complex data types.

Like records, families are collections of data components with different types. The data types of the members of a family are declared implicitly and may be changed once defined. You can access a single member of a family, manipulate an entire family as a single entity, assign one family to one another, and pass a family as a parameter to procedures and functions. You can also create an empty family, with no members, by using a DIM statement.

A family is a complex variable that is used to group related data components under a single family name. Any member of the family can be accessed by the name of the family and the name of the member, separated by a period.

Creating a Family

Each component of a family is called a *member*. To add members to a family, use the following statement:

```
FamilyName.MemberName = Value
```

where *FamilyName* is the name of the family to which the new member should be added, *MemberName* is the name of the new family member, and *value* is the valid CA-Realizer value assigned to the new family member. The value can be either a simple (scalar) data value or the name of a complex variable.

For example, the following creates a family named *FirstEmployee* with four members:

```
FirstEmployee.Name = "Mark Jackman"  
FirstEmployee.Birthday = StrToDate("10/15/65")  
FirstEmployee.ID = 24601  
FirstEmployee.Children = {"Jack", "Diane"}
```

Notice that the above example mixes string, numeric, and date-time data types, as well as scalars and arrays, as members of the family. A family is created by means of assignment statements, one member at a time.

Once created, a family variable exists until it is reassigned, or eliminated by using the CLEAR command. You can also CLEAR a single member of a family.

Referencing Members of a Family

To refer to a specific member of a family, simply use the family name, followed by a period and the member's name. For example, the following PRINT command references the *FirstName* member of the *Employee* family:

```
PRINT FirstEmployee.Name
```

Manipulating Families and Members

Family members can be manipulated in your programs in the same way as any other variable. For example, members can be used on either side in an assignment statement:

```
FirstEmployee.Payroll = FirstEmployee.Salary/12
```

An entire family can even be copied as a whole into another family:

```
NewHire = FirstEmployee
```

The *NewHire* family now contains the same members as the *FirstEmployee* family. Any previous family structure or data contents of the *NewHire* family is replaced by the *FirstEmployee* family.

Arrays of Families

When families are used as array elements, each element is a distinct family, each of which has its own distinct members. Of course you can assign the same family to different elements of the same array of families. If you do this, each element receives a copy of the original family that can be modified independently in the future.

A dynamic array of families can be created by first assigning an empty family as an element of an array. Or, as shown in the fourth line below, you can assign an existing family as one of the elements.:

```
DIM emptyfam AS FAMILY
CurrentEmployee[1] = emptyfam
CurrentEmployee[1].name = "Wolf Fliegelman"
CurrentEmployee[2] = FirstEmployee
```

Arrays of families can be expanded dynamically, but before any assignment to a member of a family, you must first assign an empty family or existing family to the element.

```
CurrentEmployee[3] = emptyfam
CurrentEmployee[3].age = 35
```

Fixed arrays of families can also be created with a DIM statement:

```
DIM PriorEmployee(1 TO 10) AS FAMILY
```

In this case, each element has already been initialized as a family, so members can be added immediately. For example, the next two statements show that the entire *FirstEmployee* family, which is currently the first element in the *CurrentEmployee* array, may be added as the fifth element of the *PriorEmployee* array and removed from the *CurrentEmployee* array.

```
PriorEmployee[5] = CurrentEmployee[1]  
CurrentEmployee[1] = emptyfam
```

Families in Nested Complex Data Structures

The previous section showed how families and arrays can be combined to create a two-level nested data structure. Families can also be used to create multi-level data structures involving more than two complex data types. CA-Realizer allows you to create extremely general data structures by using complex data types as the components of other complex data types.

These complex multi-level data structures can be pictured as trees, in which each component is represented as a lower level node connected to its (complex) parent. To access data in such a tree structure, you must specify the index numbers for any arrays, field names for any records, and member names for any families that describe the path to the desired (simple) data item.

Chapter 3

Manipulating Numbers

In This Chapter	3-1
Working with Operators, Expressions, and Functions	3-1
Operators	3-2
Mathematical Operators	3-3
Comparison Operators	3-4
Logical Operators	3-6
Bitwise Operators	3-8
Expressions	3-9
Functions	3-10
Working with Array Values	3-11
Operating on Arrays	3-11
Adjusting an Array's Valid Range	3-13
Mixing Scalars and Arrays	3-13
Subscripting Expressions	3-14
Working with Mathematical Values	3-15
Trigonometric and Exponential Functions	3-15
Rounding Functions	3-17
Random Functions	3-19
Array-Specific Functions	3-20
Mathematical Functions	3-20
Matrix Functions	3-23
Statistical Functions	3-24
Packing and Switching Functions	3-26
Time-Series Functions	3-28
Reporting Mathematical Errors	3-30

Working with Date-Time Values	3-30
Creating and Converting to Date-Time Values	3-31
Creating Date-Time Values	3-31
Converting Date-Time Values to Other Data Types	3-32
Setting the System Date and Time	3-33
Examining Date-Time Values	3-33
Manipulating Date-Time Values	3-35
Plus (+) and Minus (-) Operators	3-35
AddToTime and AddToDate	3-35

Chapter 3

Manipulating Numbers

In This Chapter

Operators, expressions, and functions are used to manipulate data. This chapter describes how to use variables, operators, and functions to form mathematical expressions.

Working with Operators, Expressions, and Functions

Operators

The CA-Realizer language uses *operators* to combine, compare, and manipulate data. An operator performs a calculation—such as addition and subtraction—or produces a logical result from a comparison.

For example, try running these sample operations:

```
Angle = 2^5
Radians = 3.141593 / 180 * Angle
PRINT Radians + 5
PRINT Angle > 30
```

Most operators are binary. That is, they take two values and produce one. For instance, you use the multiply operator (*) to multiply variables *A* and *B* by writing *A * B*. When this operation is complete, a single value (the *return value*) is produced. Therefore, *A * B* returns the product of *A* and *B*.

Expressions

An *expression* consists of one or more operators and/or functions used together to produce a value. You can assign this value to a variable or pass it to a procedure or function. For instance, the following line is an expression:

```
Abs((A + B) * C) * (3.1416/180))
```

Functions

Abs (used in the above example) is one of the CA-Realizer built-in *functions*. In general, functions encapsulate a particular process or algorithm and are often combined with operators and variables to form equations.

The above expression is evaluated when the statement in which it is contained is executed. The following statements illustrate several ways in which you can use this expression:

```
PRINT Abs(((A + B) * C) * (3.1416/180))  
x = Sin(Abs(((A + B) * C) * (3.1416/180)))  
RETURN Abs(((A + B) * C) * (3.1416/180))
```

Operators

The operators supported by CA-Realizer can be classified into the following four categories:

Category	Description
Mathematical	Perform calculations.
Comparison	Perform tests and return a value depending on whether the result is true or false.
Logical	Work on true and false values.
Bitwise	Similar to logical operators, except that the operator is applied between each pair of corresponding bits in two values.

Individually, and together, these operators allow you to perform a wide range of computations.

Mathematical Operators

Mathematical operators perform calculations. The following lists and demonstrates each mathematical operator provided by CA-Realizer:

Operator	Function	Example	Result
+	Addition	3 + 4	7
	Unary plus	+ 6	6
-	Subtraction	8 - 5	3
	Unary minus	- 9	-9
*	Multiplication	5 * 9	45
/	Division	23 / 5	4.6
^	Exponentiation	2^16	65536
Mod	Modulus (the remainder after division)	17 Mod 6	5
\	Truncated division	23 \ 5	4

Note that the plus (+) and minus (-) operators can also be used with string and date-time variables. For example, if *A* is a date and *B* is a numeric, then *A* + *B* determines the date formed by adding *B* days to *A*.

Likewise, *A* - *B* subtracts *B* days from *A*. On the other hand, if both *A* and *B* are dates, *A* - *B* returns the number of days between *A* and *B*.

Try running the following example to view how the plus (+) and minus (-) operators operate on two dates:

```
Today = QDate
Tomorrow = Today + 1
Yesterday = Today - 1
PRINT Yesterday, Today, Tomorrow
```

You can also concatenate a pair of text strings with the plus (+) operator. For example, if *A* is a string and *B* is a string, *A + B* returns a string with *B* appended to *A*.

Run the following example to view how the plus (+) operator concatenates two strings:

```
Last = "Chaplin"  
First = "Charlie "  
PRINT First + Last
```

Comparison Operators

Comparison operators compare two values and then return a value that indicates the result of the comparison. If the comparison is true, the result of the expression is 1; if the comparison is false, the result is 0.

Note that the values on both sides of a comparison operator must be the same type. CA-Realizer considers any non-zero value as true and zero as false.

The following table lists the available comparison operators:

Operator	Function	Usage
=	Equal to	<i>A</i> = <i>B</i>
<>	Not equal to	<i>A</i> <> <i>B</i>
<	Less than	<i>A</i> < <i>B</i>
<=	Less than or equal to	<i>A</i> <= <i>B</i>
>	Greater than	<i>A</i> > <i>B</i>
>=	Greater than or equal to	<i>A</i> >= <i>B</i>

The following code fragment illustrates how the comparison operators work:

```
Angle = 120
Obtuse = (Angle > 90)
Acute = (Angle < 90)
PRINT "Obtuse: "; Obtuse
PRINT "Acute: "; Acute
```

Writing *Angle > 90* asks if the value of *Angle* is greater than 90. Since *Angle* is 120, it is greater than 90 and the answer to this question is yes, or true. The *>* operator returns the value 1 to indicate a true result and 0 for a false result. Therefore, for an *Angle* of 120, the *Obtuse* variable contains 1 and the *Acute* variable contains 0.

Strings are compared alphabetically. Therefore, "about" is less than "acme", and "foot" is greater than "foo". Other characters in the string are compared according to their ASCII values.

By default, string comparisons are case-sensitive (for example, "Hello" is not equal to "HELLO"), and strings beginning with an uppercase letter are "less than" strings beginning with a lowercase letter. You can make both uppercase and lowercase equal using the SetSys command with the `_CaseSensitive` constant.. For more information, see "Manipulating Strings" in this guide and the SetSys command in the *CA-Realizer Language Reference Guide*.

Date-time values are compared as instances in time. Therefore, 5/10/93 10:30:00 is greater than both 5/10/93 09:00:00 and 5/8/93 12:00:00. Note that the date portion is more significant than the time portion. To compare only the time or date portions of date-time values, use the `_TimeNum` or `_DateNum` constants with the DateToNum function or the elements returned by the DateInfo function.

Logical Operators

Logical operators work on true and false values. You can combine them with comparison operators and other values to form a logical statement. For example:

```
Accepted = Math > 750 And Verbal > 650
UnitPrice = (Order < 500) * NormalPrice + \\  
            (Order >= 500) * BulkPrice
```

Accepted is a value of 0 or 1. Since all non-zero values in a logical statement are considered true, the three variables in the statement below are true, and the result of this operation is true:

```
RockNumber = 5
PaperNumber = 2
ScissorNumber = 8
InventoryFull = RockNumber And PaperNumber \\  
                And ScissorNumber
```

The following table lists the available logical operators:

Operator	Usage	Description
And	A And B	Returns the logical And of <i>A</i> and <i>B</i> . If both <i>A</i> and <i>B</i> are non-zero, 1 is returned; otherwise, And returns 0.
Or	A Or B	Returns the logical Or of <i>A</i> and <i>B</i> . If either <i>A</i> or <i>B</i> are non-zero, 1 is returned; otherwise, Or returns 0.
Not	Not A	Returns the logical inverse of <i>A</i> . If <i>A</i> is non-zero, 0 is returned; otherwise, Not returns 1.

Continued

Continued

Operator	Usage	Description
Xor	A Xor B	Returns the logical exclusive-Or of A and B. If A is non-zero and B is zero, or A is zero and B is non-zero, 1 is returned; otherwise, Xor returns 0.
Eqv	A Eqv B	Returns the logical equivalence of A and B. If both A and B are non-zero, or both A and B are zero, 1 is returned; otherwise, Eqv returns 0.
Imp	A Imp B	Returns the logical implication of A and B. If A is zero or B is non-zero, 1 is returned; otherwise, Imp returns 0.

This next table summarizes the results of applying each logical operator to two variables, A and B. The values of A and B change and are indicated by a T (true or a non-zero value) and F (false or 0).

A	B	And	Or	Xor	Eqv	Imp
F	F	F	F	F	T	T
F	T	F	T	T	F	T
T	F	F	T	T	F	F
T	T	T	T	F	T	T

Bitwise Operators

Bitwise operators are similar to logical operators, except that the operator is applied between each pair of corresponding bits in two values. You generally use bitwise operators to set, clear, and test the bits in a value where the individual bits or sets of bits have specific meanings.

Bitwise manipulations are handled through the functions listed in the following table. In each of these functions, the values are treated as 32-bit unsigned values. The rightmost (least significant) bit is always bit 0.

Function	Returns
<code>BitAnd(<i>n1</i>, <i>n2</i>)</code>	The bitwise AND of two values.
<code>BitOr(<i>n1</i>, <i>n2</i>)</code>	The bitwise OR of two values.
<code>BitXor(<i>n1</i>, <i>n2</i>)</code>	The bitwise exclusive OR of two values.
<code>BitNot(<i>n1</i>)</code>	The bitwise negation of a value.
<code>BitLShift(<i>n1</i>, <i>NumBits</i>)</code>	The value <i>n1</i> shifted left by <i>NumBits</i> bits.
<code>BitRShift(<i>n1</i>, <i>NumBits</i>)</code>	The value <i>n1</i> shifted right by <i>NumBits</i> bits.
<code>BitSet(<i>n1</i>, <i>BitNum</i>)</code>	The value <i>n1</i> with bit <i>BitNum</i> set to 1.
<code>BitClear(<i>n1</i>, <i>BitNum</i>)</code>	The value <i>n1</i> with bit <i>BitNum</i> set to 0.
<code>BitTest(<i>n1</i>, <i>BitNum</i>)</code>	The value of bit <i>BitNum</i> in <i>n1</i> .

Expressions

An expression consists of one or more operators and/or functions, which produce a value. This value can be assigned to a variable or passed to a procedure or function.

Expressions are evaluated from left to right according to the order of operations. When you use two or more operators to compute a single value, there is a specific order in which operations occur because certain operators have precedence over others.

This precedence is defined in the following table:

Operator Symbols	Operator Names
()	Parentheses
[]	Array brackets
^	Exponentiation
- (unary minus)	Unary minus
* /	Multiplication and division
\	Truncated division
Mod	Modulus
+ -	Addition and subtraction
= <> > < >= <=	Comparison
Not	Logical negation
And	Logical And
Or	Logical Or
Xor	Logical Exclusive Or
Eqv	Logical Equivalence
Imp	Logical Implication

For example, in the equation $X + Y * Z$, Y and Z are first multiplied, and then X is added to the product because multiplication always has a higher precedence than addition. Whenever an equation combines operators of equal precedence, the operations are performed from left to right.

Parentheses can be used to override the order of operations, since the operations within parentheses are performed first. For example, using the same equation as above, you can force the addition of $X + Y$ to be performed first by entering $(X + Y) * Z$. In this case, the sum of $X + Y$ is multiplied by Z .

Functions

In general, *functions* encapsulate a particular process or algorithm. You can combine them with operators and variables to form expressions.

CA-Realizer functions provide a wide range of capabilities and can be used with all different types of data to perform a number of complex calculations, such as mathematical computation, date-time manipulations, time series analysis, statistical analysis, and string manipulation. An additional feature of the CA-Realizer functions is that most work equally well with arrays and scalars.

Functions, like commands, are often given one or more values (called *parameters* or *arguments*), and like operators, return a single array or scalar as a result.

Functions are also similar to variables in that they can also be used in equations. For example, you can use the `QDate` function to obtain the current date in either of the following ways:

```
Today = QDate  
PRINT Today
```

or

```
PRINT QDate
```

Note however, that unlike variables, functions cannot appear on the left side of the assignment operator (=).

Note: Because manipulating strings is such an intrinsic part of BASIC programming, we have devoted an entire chapter to this topic. See “Manipulating Strings” for more information. In addition, see “Structured Programming with Procedures and Functions” to learn how to create your own functions.

Working with Array Values

To facilitate working with array values, CA-Realizer provides enhancements to conventional BASIC array-handling techniques.

Operating on Arrays

Multiplication

One of the most powerful features in CA-Realizer is the ability to process entire arrays without having to run through a loop. For example, to multiply two arrays (*A* and *B*, each of which contains 8000 elements) in conventional BASIC, the following loop would be used:

```
FOR i = 1 to 8000
    F(i) = A(i) * B(i)
NEXT i
```

However, CA-Realizer obtains the same result using a single statement:

```
F = A * B
```

This latter approach is much faster and more convenient than using loops. For example, assume an organization's sales data contains a price and a quantity sold for each item in their inventory. The value of sales for each item is determined by multiplying the *Price* array by the *Quantity* array. You can do this for all items by multiplying the two arrays as shown below:

```
SaleValue = Price * Quantity
PRINT SaleValue
```

Comparison and
Logical

The multiplication (*) operator applies to each corresponding element in the *Price* and *Quantity* arrays. *Price*[1] is multiplied by *Quantity*[1], *Price*[2] by *Quantity*[2], and so on. The result is contained in an array called *SaleValue*.

The comparison and logical operators also work with arrays, yielding an array of 1's and 0's. For example, if *A* and *B* still contain 8000 elements each,

`F = A < B`

produces an array with 8000 elements, each of which is 1 if the corresponding elements in *A* is less than the one in *B*, and 0 if it is not.

There are powerful ways to apply comparison and logical operators. For example, the following statements determine the number of products priced within a certain range:

```
MinPrice = 3.00
MaxPrice = 10.00
InRange = Price > MinPrice And Price < MaxPrice
PRINT Sum(InRange)
```

The logical statement in this example builds an array called *InRange*, which contains 1's for each corresponding element in the *Price* array greater than 3.00 and less than 10.00. The *Sum* function adds the 1's and obtains the total number of *Price* elements which satisfy the condition.

This built-in array processing makes manipulating data with CA-Realizer very powerful.

Adjusting an Array's Valid Range

If the valid range of arrays used in calculations do not match, CA-Realizer only applies operators to those elements in which the two array ranges overlap. For example, if array *A* has a range of 1 to 10, and array *B* has a range of 21 to 30, the operation $A * B$ does not work.

To accomplish this calculation, use the two *shift operators* to adjust the valid range of an array. For example, shift array *A* ahead 20 elements using $A[-20]$, or shift array *B* back 20 elements using $B[+20]$. Both of the following operations produce a valid result:

```
R1 = A[-20] * B
R2 = A * B[+20]
```

The valid range for the resulting array *R1* is 21 to 30 and the valid range for *R2* is 1 to 10.

Mixing Scalars and Arrays

You can mix scalars and arrays in these operations as well. Try multiplying the entire *Price* array by the *ProductNumber* variable using the equation $SaleValue = Price * ProductNumber$. Then compute the daily average of *SaleValue* with the equation $Avg = SaleValue / 7$.

The rules for determining the results of mixing arrays and scalars in computations are as follows:

- If both values are scalars, the result is a scalar.
- If one value is a scalar and the other is an array, the operator is applied between the scalar and each element of the array. The result is an array with the same valid range as the original array.
- If both values are arrays, the operator is applied between elements with the same indices. The result is an array valid over the intersection of the valid ranges. An error is produced if the valid ranges do not intersect.

Subscripting Expressions

To subscript and shift arrays, use the bracket operators (`[]`). You can also apply these bracket operators to expressions that produce arrays.

For example, if *A* and *B* are both arrays, the following statement multiplies each element in *A* by the corresponding element in *B*, and then assigns elements 11 through 20 from the result to *F*:

```
F = (A * B)[11:20]
```

F will be an array of size 10, valid from 11 to 20.

Expression subscripting is useful with functions that return an array, and only a part of the array is needed. For example, `ChartQ` is a CA-Realizer function that returns a four-element array representing the size and position of the current chart. If your program requires only two of the elements in the returned array, you could execute the following statement:

```
ChartSize[1:2] = ChartQ(_Size)[3:4]
```

This statement assigns only elements 3 and 4 (the width and height of the chart) of the returned array to the array *ChartSize*.

Working with Mathematical Values

CA-Realizer provides a complete set of functions that enable you to work with and manipulate mathematical values. Functions in this group generally fall into the following categories:

- Trigonometric and exponential
- Rounding
- Random
- Array-Specific
 - Mathematical
 - Matrix
 - Statistical
 - Packing and switching
 - Time-series

This section also describes how CA-Realizer reports mathematical errors.

Trigonometric and Exponential Functions

The CA-Realizer language includes a large set of common mathematical functions for trigonometric and exponentiation calculations. These functions are summarized below:

Function	Description
Cos	Computes the cosine of a value given in radians.
Sin	Computes the sine of a value given in radians.
Tan	Computes the tangent of a value given in radians.
ACos	Returns the arc cosine of a value in radians.
ASin	Returns the arc sine of a value in radians.

Continued

Continued

Function	Description
ATn	Returns the arc tangent of a value in radians.
ATan2	Computes the tangent of a line from the origin to a point.
Exp	Computes e^x .
Exp10	Computes 10^x .
Log	Computes $\ln(x)$.
Log10	Computes $\log_{10}x$.
Sqr	Computes x^2 .
Sqrt	Computes the square root of a value.

Trigonometric

All trigonometric functions require that you supply a real number representing the angle to be computed.

Angle measurements are in radians. Degree measurements are converted to radians by multiplying by the factor $\pi / 180$.

To convert from radians to degrees, do the following:

```
Degrees = 60  
Radians = Degrees * (_Pi / 180)  
PRINT Cos(Radians)
```

For example, the following code creates an array with the value of Sin for 1 to 360 degrees:

```
s = Sin(Index(360) * (_Pi/180))
```

Exponential

Exp and Exp10 compute e^x and 10^x , respectively. Their inverses, Log and Log10, compute $\ln(x)$ and $\log_{10}x$, respectively.

This fact can be used to compute the logarithm of a number for a base other than e or 10 by using the following formula:

$$\text{Log}_b x = \text{Log}(x) / \text{Log}(b)$$

To calculate the value of the constant e , use the following equation:

$$e = \text{Exp}(1)$$

Sqr and Sqrt compute the square and square root of x , respectively. To find the cube root of a number, raise it to the $1/3$ power with the $^$ operator (the $^$ operator computes x^y).

This can be generalized easily to the n^{th} root of x with the following formula:

$$n^{\text{th}} \text{ root of } x = x ^ (1/n)$$

Rounding Functions

Rounding functions can be used to adjust data in a number of ways. The full set of rounding functions are presented below:

Functions	Description
Round	Returns the nearest integer to a value.
Ceil	Returns the smallest integral number greater than or equal to a value.
Floor or INT	Returns the largest integral number less than or equal to a value (Floor and INT are synonyms).
Fix	Returns a value rounded toward zero.
Abs	Returns the absolute value of a number.
Sgn	Returns the sign function of a number.
Bool	Converts a number to a Boolean value, either 1 or 0.

The following example demonstrates the differences between the Round, Ceil, and Floor functions by manipulating the numbers in *MyData*:

```
MyData = {1.0, 3.25, 5.75, -3.75, -6.50}
PRINT "Number", "Round", "Ceil", "Floor"
PRINT MyData, Round(MyData), Ceil(MyData), \
      Floor(MyData)
```

When you run this example, the following information is displayed in the Print Log:

<i>Number</i>	<i>Round</i>	<i>Ceil</i>	<i>Floor</i>
1.0000	1.0000	1.0000	1.0000
3.2500	3.0000	4.0000	3.0000
5.7500	6.0000	6.0000	5.0000
3.7500	4.0000	3.0000	4.0000
6.5000	7.0000	6.0000	7.0000

Round

The Round function returns the nearest integer value of a specified number. If a number falls exactly between two integer values with a fractional part of .5, the number is rounded away from zero. Thus, Round(3.5) returns 4.0, and Round(-3.5) returns -4.0.

Ceil and Floor

Ceil and Floor functions return numbers rounded in a specific way. The Ceil function returns the minimum integer number greater than or equal to a value, while the Floor function returns the maximum integer number less than or equal to a value.

Random Functions

CA-Realizer provides two functions to generate pseudo-random numbers, `Rnd` and `RANDOMIZE`.

`Rnd`

The `Rnd` function generates an approximately uniform random number between 0.0 and 1.0. `Rnd` is generally used as follows:

```
RandomValue = Rnd [(ArraySize)]
```

If *ArraySize* is given, it generates an *ArraySize* length array of random numbers.

To generate random numbers between 0 and *N*, multiply the results by *N*. For example, `Rnd * 10` generates a random number between 0 and 10. You can use the rounding functions to generate random integer values.

`RANDOMIZE`

`RANDOMIZE` reseeds the random number generator. This is useful for reproducing or varying the set of random numbers produced.

`RANDOMIZE` is generally used as follows:

```
RANDOMIZE Seed
```

The series of random numbers produced after seeding with a given seed does not vary. To regenerate a particular set of random numbers, use `RANDOMIZE` with the same seed. Call `RANDOMIZE` with a different seed to create a different set of random numbers.

To obtain a different set of random numbers each time a program is run, you can reseed the random number generator with a value related to the current time, such as the number of seconds since midnight:

```
RANDOMIZE TIMER
```

Array-Specific Functions

CA-Realizer contains a set of array-related functions that can be used to manipulate arrays.

Mathematical Functions

While all mathematical functions work with arrays, some array-specific mathematical functions are available. These functions are listed below:

Function	Description
StartValid	Returns the starting index of an array's valid range.
EndValid	Returns the ending index of an array's valid range.
Index	Creates an array of elements set to their index value.
VectSet	Creates an array filled with 1's and 0's.
Mask	Creates an array of values set where the mask is non-zero.
ZeroFill	Replaces all 0 entries with the value of the most recent non-zero element.
RunTot	Creates an array whose elements contain the running total of the input array elements.
Sum	Returns the sum of all the elements in an array.
Product	Returns the product of all the elements in an array.
Min	Returns the minimum of a set of numbers.
Max	Returns the maximum of a set of numbers.

Index and VectSet

Both the Index and VectSet functions can be used to create and fill a new array. The Index function creates an array of a given size and assigns each element its index value. The first element gets the value 1, the second gets the value 2, and so on. Run the following example to see how Index works:

```
Series = Index(90)
PRINT Series

NewSeries[1:10,1:9] = Series
PRINT NewSeries
```

The following example uses Index with date-time values to create a list of the next fifteen days. Index is also used to fill an array with the values of 1 to 90 degrees, converted to radians:

```
PRINT QDate + Index(15)

Radians = _Pi / 180 * Index(90)
PRINT Radians, Sin(Radians)
```

The VectSet function creates an array filled with 1's and 0's. VectSet expects three parameters that define the range and interval over which to place the 1's. For example, specify a 20-element array by running the following lines:

```
Pattern = VectSet(10, 3, 20) * 6
PRINT pattern
```

This sets every third element to 6, starting with the tenth element.

Sum, Product, Min, and Max

There are also functions that can be used to compute the sum or product of elements in an array. Use the Sum function to find the total of all the elements in an array. Run the following lines to apply the Sum function to the sales report data in the *Quantity* array:

```
Quantity = {25, 33, 6, 10, 7}
TotalSales = Sum(Quantity)
PRINT TotalSales
```

Likewise, the Product function returns the value of all elements multiplied together.

The Min and Max return the highest and lowest values from a set of numbers. These two functions operate on various combinations of number sets. Min and Max take parameters in an identical fashion. For example, try the following line:

```
PRINT Min(10,15,22,33,5), Max(10,15,22,33,5)
```

Given a list of scalars, Min returns the value of the smallest and Max returns the value of the largest. In the statement above, Min returns 5 and Max returns 33. When passed an array, Min and Max return the smallest and largest element in the array:

```
MaxSale = Max(Quantity)
MinSale = Min(Quantity)
PRINT "Largest quantity = ", MaxSale
PRINT "Smallest quantity = ", MinSale
```

When scalars and arrays are mixed using Min and Max, the functions return an array of the element-by-element minimum or maximum for the elements in the intersection of valid ranges.

StartValid and
EndValid

The StartValid and EndValid functions return information about an array's valid range. StartValid returns the first index in an array's valid range, and EndValid returns the last index in an array's valid range.

Use these functions together to determine the size of an array. For example:

```
First = StartValid(Quantity)
Last = EndValid(Quantity)
PRINT "Size = ", Last - First + 1
```

The StartValid and EndValid functions take an optional second *DimNum* parameter which allows you to specify the dimension over which they are to operate. For example, to determine the lengths of dimensions 1 and 2 in the *NewSeries* array respectively, use:

```
EndDim1 = EndValid(NewSeries, 1)
EndDim2 = EndValid(NewSeries, 2)
```

Matrix Functions

CA-Realizer provides three matrix functions that address the most commonly used matrix manipulations.

MatTranspose

MatTranspose returns the transposition of the given matrix. In other words, if the matrix is $N \times M$, the result is $M \times N$.

For example:

```
A[1:3,1:3] = Index(9)
B = MatTranspose(A)
PRINT A
PRINT B
1  2  3
4  5  6
7  8  9
1.0000  4.0000  7.0000
2.0000  5.0000  8.0000
3.0000  6.0000  9.0000
```

MatInvert

MatInvert returns the inversion of the given *square* matrix, which is the matrix that, when multiplied by the original, yields the identity matrix.

Note: It is not always possible to invert a matrix.

Try the following:

```
C[1,1:3]={1,3,2}
C[2,1:3]={3,3,5}
C[3,1:3]={5,9,1}

D = MatInvert(C)
PRINT C
PRINT D
1  3  2
3  3  5
5  9  1
-0.8750  0.3125  0.1875
0.4583  -0.1875  0.0208
0.2500  0.1250  -0.1250
```

MatMult

MatMult returns the matrix multiplication of two given matrixes. If *Matrix1* is an $N \times M$ matrix, and *Matrix2* is an $M \times P$ matrix, the result is an $N \times P$ matrix. For example:

```
E = MatMult(A, B)
PRINT A, B
PRINT
PRINT E
1  2  3  1.0000  4.0000  7.0000
4  5  6  2.0000  5.0000  8.0000
7  8  9  3.0000  6.0000  9.0000
14.0000  2.0000  50.0000
32.0000  77.0000  122.0000
50.0000  122.0000  194.0000
```

Statistical Functions

The statistical functions provided by CA-Realizer form another set of powerful array functions. (All the functions in this category have names that begin with the “Stat” prefix.)

The available statistical functions are listed below:

Function	Description
StatNorm	Normalizes an array to a specified range.
StatPercent	Returns values as a percentage of the sum of all the array values.
StatRank	Determines the rank of elements in an array.
StatSort	Sorts an array in ascending order.
StatRegress	Creates a “best fit” least-squares regression.

StatSort

One of the most commonly used Stat functions is StatSort, which sorts an array in ascending order. StatSort can be passed one or two arrays. If StatSort is given only one array, it sorts and returns a copy of that array.

```
Quantity = {25, 33, 6, 10, 7}
PRINT StatSort(Quantity)
```


If `StatSort` is called with two arrays, it sorts the elements of the first array, then arranges the corresponding elements of the second array according to the order of the first. Run the following example to see how this works:

```
Name = {"rock", "round stone", "square stone", \\  
        "metal scissors", "heavy paper"}  
Price = {2.30, 3.75, 10.50, 1.25, 7.75}  
PRINT StatSort(Quantity, Name)  
PRINT StatSort(Price, Name)
```

These statements produce two lists of names. The first list is sorted by increasing *Quantity* and the second list by increasing *Price*.

`StatNorm`,
`StatPercent`, and
`StatRank`

These three `Stat` functions scale data in an array. `StatNorm` *normalizes* an array to a specified range. This means that the smallest element in the array is mapped to the bottom of the range, the largest element is mapped to the top of the range, and all intermediate values are linearly scaled.

`StatPercent` computes the sum of all elements in an array and returns an array containing the input array divided by the sum. The value of each element is converted into a percentage of the whole array.

`StatRank` determines the rank of elements in an array. The *rank* of an element is its position, if the array was sorted in ascending order.

Apply these functions to the *Quantity* array to see how they perform:

```
PRINT "StatNorm", "StatPercent", "Quantity", \\  
      "StatRank", "StatSort"  
PRINT StatNorm(Quantity, 0, 1), \\  
      StatPercent(Quantity), Quantity, \\  
      StatRank(Quantity), StatSort(Quantity)  
  
StatNorm StatPercent Quantity StatRank StatSort  
0.7037   0.3086      25.0000   4.0000   6.0000  
1.0000   0.4074      33.0000   5.0000   7.0000  
0.0000   0.0741       6.0000   1.0000  10.0000  
0.1481   0.1235     10.0000   3.0000  25.0000  
0.0370   0.0864       7.0000   2.0000  33.0000
```

Notice that `StatNorm` scales the numbers in the *Quantity* array to fit between 0 and 1.

Also, look at the result of StatRank when it is applied to *Quantity*, and compare the sorted result of StatSort with the rank values given by StatRank.

When two or more elements in an array have the same value, their rank is shared. The equation $M + 1/N$ (N being equal numbers that are the M^{th} largest element) computes the rank for matching values. For example, if an array contains {5, 3, 10, 3, 4}, the result is {4, 1.5, 5, 1.5, 3}.

Packing and Switching Functions

The following table lists the CA-Realizer array packing/switching functions; these functions allow you to compress, manipulate, or restore arrays:

Function	Description
StatPack	Compresses (or packs) an array.
StatUnpack	Restores a packed array.
StatSwitch	Creates an array from two Boolean arrays.

Use the StatPack and StatUnpack functions to compress and restore arrays. These functions reduce the storage associated with *sparse* arrays (that is, arrays which contain many 0's).

StatPack

When StatPack compresses a sparse array, it maintains original position information; you can also use it with logical operators to return information about elements satisfying a certain condition.

StatPack is generally used as follows:

```
StatPack(Result, Switch, Data)
```

where *Switch* is a Boolean array indicating the data to be pulled out of the array specified by *Data*. *Result* is the name of the family into which CA-Realizer should place the results.

CA-Realizer creates a member named *PV* (Packed Value) in the family specified by *Result* that contains all elements for which *Switch* is non-zero. It then creates a member named *PI* (Packed Index), which contains the numbers of the elements to which these originally belonged. Two other members—*PSV* (Packed Start Valid) and *PEV* (Packed End Valid)—indicate the starting and ending valid range of the original array.

StatUnpack

StatUnpack restores a packed array. It can take a family created by StatPack, or it can take the individual members separately. For example:

```
ReturnArray = StatUnpack(Packed)
ReturnArray = StatUnpack(Index, Data
                        [, StartValid [, EndValid]])
```

In the second form, *Index* is an array of indices and *Data* is a data array. CA-Realizer places each element in the data array in an element corresponding to the number in the index array. The remaining elements are default values.

Note: If you do not specify *StartValid* and *EndValid* with StatUnpack, the resulting array is valid from 1 until the last value in *Index*.

StatSwitch

The StatSwitch function creates a Boolean array of 1's and 0's based on two other arrays:

```
ReturnArray = StatSwitch(On, Off)
```

If element *i* in both *On* and *Off* are 0, element *i* in the resulting array is the same as element *i-1*. If element *i* in *On* is non-zero, element *i* in the resulting array is set to 1. If element *i* in *Off* is non-zero, element *i* in the resulting array is set to 0. If element *i* is non-zero in both *Off* and *On*, element *i* in the resulting array is set to 0.

Time-Series Functions

There are several functions which allow you to perform operations on an array's subranges. Several of these functions perform an operation—such as computing an average—that involve an element and the preceding N elements. You can define N as a constant or change it for each element.

The available time-series functions are listed below:

Function	Description
StatMovAvg	Determines a moving average for an array.
StatMovMax	Determines a moving maximum for an array.
StatMovMin	Determines a moving minimum for an array.
StatMovTot	Determines a moving total for an array.

Each of these functions takes the same two arguments *Data* and *Lengths*, where *Data* represents an array and *Lengths* is a scalar or an array representing the values of N .

```
ReturnArray = StatMovAvg(Data, Lengths)
```

StatMovAvg determines the moving average of an array. The average of the i^{th} data element and the preceding $N-1$ elements (where N is the i^{th} element of *Lengths*) results in the i^{th} element. If *Lengths* is a scalar, N is always equal to *Lengths*.

Because the number of elements over which the function is computed can change on an element-by-element basis (and lengths can specify values that extend beyond the range of the array), it is possible for undefined values to occur in the middle of an array. For example, suppose N is 15 for the fourth element of the array. This would mean that the fourth element is a function of the preceding 15 elements.

This clearly extends beyond the range of the array, so the fourth element of the result is undefined. When an undefined value occurs in the middle of the result array, all previous values are considered undefined as well. Therefore, the valid range of the array is reduced every time an undefined value is encountered.

For example, declare the following arrays:

```
Values = {1, 5, 8, 4, 9, 6, 4, 2, 8, 3}
Lengths = {1, 2, 2, 2, 2, 3, 3, 2, 2, 4}
```

The StatMovAvg, StatMovMax, StatMovMin, and StatMovTot commands produce the following results when they use *Values* and *Lengths* as their arguments. Each example first defines *N* as a constant and then uses *Lengths* to change the value of *N* for each element.

```
StatMovAvg(Values, 3) = \\  
  {4.67, 5.67, 7, 6.33, 6.33, 4, 4.67, 4.33}  
StatMovAvg(Values, Lengths) = \\  
  {1, 3, 6.5, 6, 6.5, 6.33, 6.33, 3, 5, 4.25}  
StatMovMax(Values, 3) = \\  
  {8, 8, 9, 9, 9, 6, 8, 8}  
StatMovMax(Values, Lengths) = \\  
  {1, 5, 8, 8, 9, 9, 9, 4, 8, 8}  
StatMovMin(Values, 3) = \\  
  {1, 4, 4, 4, 4, 2, 2, 2}  
StatMovMin(Values, Lengths) = \\  
  {1, 1, 5, 4, 4, 4, 4, 2, 2, 2}  
StatMovTot(Values, 3) = \\  
  {14, 17, 21, 19, 19, 12, 14, 13}  
StatMovTot(Values, Lengths) = \\  
  {1, 6, 13, 12, 13, 19, 19, 6, 10, 17}
```

Reporting Mathematical Errors

By default, if a function results in a mathematical error, such as dividing a number by zero or taking the square root of a negative number, CA-Realizer returns an appropriate value to indicate the nature of the error as follows:

Error Value	Meaning
1.#INF	Infinite value
1.#QIND	Indefinite value
1.#QNAN	Not a number

You can instruct CA-Realizer to stop and display an error whenever an invalid mathematical operation occurs by setting the `_FPError` system variable with the `SetSys` command:

```
SetSys(_FPError, 1)
```

Working with Date-Time Values

Date-time functions are used to process date-time values, each of which represents a particular second in time. A date-time value is a data type used to store a date and time. Both the date and time values are stored together in the single date-time variable, although the date and time portions can be displayed and manipulated separately.

The following table lists the CA-Realizer date-time functions:

Function	Description
DayOfWeek	Returns the day of the week for a date.
DateToNum	Converts a date-time value to a number.
DateInfo	Returns information about a date-time value.

Continued

Continued

Function	Description
QDate	Returns the current date and time.
NumToDate	Converts a real value to a date-time value.
StrToDate	Converts a string to a date-time value.
AddToTime	Adds hours, minutes, and seconds to a date-time value.
AddToDate	Adds years, months, days, hours, minutes, and seconds to a date-time value.
DATE\$	Returns the current date as a string.
TIME\$	Returns the current time as a string.
TIMER	Returns the number of seconds that have passed since midnight.

Note: For a complete list of valid date-time formats, see the PRINT command in the *CA-Realizer Language Reference Guide*.

Creating and Converting to Date-Time Values

The section describes the CA-Realizer functions that can be used to create and convert date-time values.

Creating Date-Time Values

To create/generate date-time values, use the following functions:

QDate	The QDate function returns the current date and time from your computer's system clock. For example: <pre>Today = QDate PRINT Today</pre>
TIMER	The TIMER function returns the number of seconds that have passed since midnight.

StrToDate

The StrToDate function converts a text string that contains a standard date and time format to a date-time value. For example:

```
PRINT StrToDate("11-Nov-68")
StartDate = StrToDate("10/25/90")
StartTime = StrToDate("01:10:24 AM")
PRINT StartDate, StartTime
```

The following results are displayed:

```
11/11/68
10/25/90  01/01/00
```

Since the date portion of *StartTime* in the example above has not been set, the date printed is the default system date January 1, 1900. Furthermore, notice that PRINT only prints the date portion of a date-time variable and ignores the time portion.

Note: In order to have access to all the information stored in a date-time value, use the special date-time functions or issue the PRINT USING command with the proper date-time formats (as discussed later in this guide).

NumToDate

Use NumToDate to convert a real value to a date-time value. It takes a date number and a time number, such as those returned by DateToNum, and creates an appropriate date-time value.

Converting Date-Time Values to Other Data Types

To convert existing date-time values to other data types, use:

DATE\$

The DATE\$ function returns the current date as a string in the form "mm-dd-yyyy" (such as "07-25-1968").

TIMES

The TIMES function returns the current time as a string in the form "hh:mm:ss" (for example, "15:10:00").

Sprint

The Sprint function converts a date-time value to a string in any format.

DateToNum

The DateToNum function converts a date-time value to a number.

Setting the System Date and Time

You can set the system date and time in two ways. The `SetSystemTime` and `SetSystemDate` commands take a date-time value and extract the corresponding portion to set the clock. The `TIME$` and `DATE$` commands set the time or date specified as a string in any of the legal `StrToDate` formats.

For example, the following are equivalent:

```
ds = StrToDate("12:05 AM 07/25/68")
SetSystemTime(ds)
SetSystemDate(ds)

TIME$ = "12:05 am"
DATE$ = "07/25/68"
```

Examining Date-Time Values

CA-Realizer provides a set of functions that allow you to find out specific information about date-time values (for example, the day of the week of a given date-time value). These functions—`DayOfWeek`, `DateToNum`, and `DateInfo`—are described below.

DayOfWeek

`DayOfWeek` indicates the day of the week on which a particular date falls. This function returns a number from 1 to 7, where Monday is represented by 1, Tuesday by 2, and so on.

Use the `DayOfWeek` and `QDate` functions together with the `Index` function to build a simple calendar for the next 30 days:

```
Dates = QDate + Index(30)
PRINT DayOfWeek(Dates), Dates
```

DateToNum

`DateToNum` converts a date-time value to a real number representing either the date portion or the time portion of the date-time value:

```
ng = DateToNum(Date, _DateNum)
ng = DateToNum(Date, _TimeNum)
```

`_DateNum` returns a real value representing the date number, where 1/1/1900 is number 2; `_TimeNum` returns the time number, which is the number of seconds since midnight.

Note: The values returned by `DateToNum` can be passed to `NumToDate` to convert them back to a date-time value.

DateInfo

`DateInfo` is a general purpose function that returns an array containing specific information about the individual values in a given date-time value. Each element in the returned array is indexed by a constant, named according to the information that particular element contains.

These constants are listed in the following table; it also describes the contents of the element with which each constant is associated:

Constant	Contents
<code>_DI_Year</code>	The full year (i.e., 1993).
<code>_DI_Month</code>	The number of the month (such as 4 for April).
<code>_DI_DayOfMonth</code>	The date within the month (like 8 for the 8 th).
<code>_DI_DayOfYear</code>	The day within the year since January 1 (i.e., 99 for April 8 th (non-leap year)).
<code>_DI_DateNum</code>	The date number, as returned by <code>DateToNum</code> (i.e., 33336 for 4/8/93).
<code>_DI_DayOfWeek</code>	The day of the week, as returned by <code>DayOfWeek</code> (i.e., 1 for Mon, Apr 8 th).
<code>_DI_Hour</code>	The hour of the day, from 0 to 23 (i.e., 14 for 2:34 PM).
<code>_DI_Minute</code>	The minute of the hour, from 0 to 59 (i.e., 34 for 2:34 PM).
<code>_DI_Second</code>	The second of the minute, from 0 to 59 (i.e., 56 for 2:34:56 PM).
<code>_DI_TimeNum</code>	The time number, as returned by <code>DateToNum</code> (i.e., 52440 for 2:34 PM).

For example, use `DateInfo` to look at the time portion of a date-time variable such as *Today*:

```
Today = QDate  
Info = DateInfo(Today)  
PRINT "Hour:", Info[_DI_Hour]  
PRINT "Minute:", Info[_DI_Minute]
```

Manipulating Date-Time Values

The plus (+) and minus (-) operators, and the `AddToTime` and `AddToDate` functions, provide two convenient ways to manipulate date-time values.

Plus (+) and Minus (-) Operators

You can use the plus (+) and minus (-) operators to manipulate date-time values as follows:

- Add a real value to a date-time value with the plus (+) operator to increase the date-time by that many days.
- Subtract a number from a date-time value with the minus (-) operator to decrease the date-time by the given number of days.
- Subtract one date-time value from another with the minus (-) operator to find the number of days between two dates.
- Add a set number of days to the date-time variable with the plus (+) operator to go to a new date.

For example, add 30 days to a date-time variable named `StartDate`:

```
PRINT StartDate + 30
```

AddToTime and AddToDate

When using the plus (+) and minus (-) operators with date-times, you are limited to computing values in terms of days. However, with the `AddToTime` and `AddToDate` functions, you can add or subtract years, months, days, hours, minutes, and seconds to or from a date-time value.

`AddToDate` adjusts a date-time variable in terms of time and date. `AddToTime` changes a date-time variable only in terms of time, although the date may change as a consequence.

When you use `AddToDate` to add a month to October 25, 1992, the `AddToDate` function knows to add 31 days, since there are 31 days in October. Here is an example of how to use `AddToDate` to add a month to a date-time variable:

```
EndDate = AddToDate(StartDate, 0, 1)
PRINT EndDate
```

The `AddToDate` function takes between two and seven parameters. The first parameter is the date-time variable to be modified. The second through seventh parameters correspond to the number of years, months, days, hours, minutes, and seconds that you want to add.

For example, the first line below adds 0 years, 1 month and 5 days to *StartDate*. The second line adds 50 minutes (1 hour - 10 minutes) and 15 seconds to *StartTime* and stores the result in *EndTime*:

```
PRINT AddToDate(StartDate, 0, 1, 5)
EndTime = AddToDate(StartTime, 0, 0, 0, 1, -10, 15)
```

Since only the time portion is important to the *StartTime* and *EndTime* variables, you can use the `AddToTime` function instead. `AddToTime` takes between two and four parameters. The first parameter indicates the time variable to be modified. The second through fourth parameters correspond to the number of hours, minutes and seconds that you want to add.

The following line adds 1 hour, 10 minutes, and 12 seconds to *StartTime*:

```
EndTime = AddToTime(StartTime, 1, 10, 12)
```

Chapter 4

Manipulating Strings

In This Chapter	4-1
CA-Realizer String Basics	4-1
Converting Between Text and ASCII Codes	4-2
Combining Strings	4-4
Comparing Strings	4-4
Retrieving Parts of a String	4-6
Retrieving from the Left, Right, or Middle of a String	4-6
Retrieving from Any Position in the String	4-8
Searching with the InStr Command	4-8
Searching with the StrTok Command	4-10
Replacing Text Within Strings	4-12
Changing Case	4-14
Inserting and Deleting Portions of a String	4-15
Converting Strings	4-16
Numbers to Strings	4-16
Strings to Numbers and Dates	4-17
Converting from External Formats	4-20
Formatting Strings	4-21
The Sprint Function	4-21
Format Codes	4-22

Chapter 4

Manipulating Strings

In This Chapter

CA-Realizer features 40 string functions that allow you to manipulate strings in a wide variety of ways.

In this chapter, you will learn how to work with text using the various string-handling functions. In addition, several of the standard CA-Realizer operators that can also be used to manipulate text data are presented.

CA-Realizer String Basics

Each individual character that is in a string—or displayed on the screen—corresponds to an integer value ranging from 0 to 255. These standard values, used on all personal computers, are called *ASCII codes*.

For example, the ASCII codes for “A”, “B”, “a” and a carriage return are 65, 66, 97, and 13, respectively. A complete list of the ASCII values for each character is provided in Appendix C of this guide.

Some of the most common manipulations involve conversions between numbers and text—and ASCII codes and text—which is the subject of the next section.

Converting Between Text and ASCII Codes

To facilitate converting strings between text and ASCII code formats, CA-Realizer provides the following functions:

Function	Description
Asc	Returns the ASCII value of the first character in a string. (Asc is the counterpart of Chr\$.)
Chr\$	Returns the character corresponding to a particular ASCII value. (Chr\$ is the counterpart of Asc.)
Len	Returns the length of a string.
String\$	Forms a string by repeating given characters.
Space\$	Returns a string containing a given number of spaces. (Spc is a synonym for Space\$ and is included for compatibility with earlier versions of BASIC.)
Reverse\$	Returns a copy of a given string with the characters reversed.

Asc and Chr\$

Asc and Chr\$ are often used together to manipulate individual characters within strings. Asc returns the ASCII value of the first character in a string. Conversely, Chr\$ returns the character corresponding to a particular ASCII value.

```
ASCIIValue = ASC(Character)
Character = Chr$(ASCIIValue)
```

The following statement returns the ASCII value for the first character in the string "bing":

```
PRINT Asc("bing")
98
```

The following statement returns the character for the ASCII value 98:

```
PRINT Chr$(98)
b
```


Len and String\$

Another useful pair of functions are Len and String\$. Len returns the length of a string, while String\$ forms a string by repeating characters.

```
Value = Len(String)
NewString = String$(Length, FillerText)
```

You can use String\$ to create filler strings when printing titles. Define the length of the string by *Length*. If *FillerText* is an integer, the string is filled with the corresponding ASCII character. (For example, String\$(Length, 13) creates a string of length *Length* filled with carriage returns.) If *FillerText* is a string, CA-Realizer repeats that string until the return string is *Length* characters long.

Note: This is different than some versions of BASIC, which simply repeat the first character from the string.

The following example prints the characters corresponding to the ASCII values 65 and 97, the ASCII value of the first character (B) in "Boing", the value of MyString, and its length:

```
MyString = String$(10, "Abc")
PRINT Chr$(65), Chr$(97), Asc("Boing"), \
      MyString, Len(MyString)
A   a   66 AbcAbcAbcA   10
```

Space\$ and Reverse\$

Two other convenient BASIC string functions are Space\$ and Reverse\$. Space\$ returns a string containing *Length* space characters (ASCII 32); Reverse\$ returns a copy of *String* with the characters reversed.

```
NewString = Space$(Length)
NewString = Reverse$(String)
```

The following example prints the reverse of the string "Randolph", adds five spaces, and then prints the reverse of the string "WasItACatISaw":

```
PRINT Reverse$("Randolph") + Space$(5) + \
      Reverse$("WasItACatISaw")
hplodnaR      waSiTaCaTiSaw
```

Combining Strings

The process of combining strings is called *concatenation*. You can combine two or more strings with the plus (+) operator. When strings are combined, no spaces are added between them.

For example:

```
A = "CA-Rea"  
B = "lizer"  
PRINT A + B  
CA-Realizer
```

Note: If you concatenate fixed-length strings, you may find more spaces than you expect, since fixed string values are left-justified and may be padded with blanks.

Comparing Strings

The following table lists the string comparison operators provided by CA-Realizer:

Operator	Description
<	Less than
<=	Less than or equal to
<>	Not equal to
=	Equal to
>	Greater than
>=	Greater than or equal to

When comparing two strings, CA-Realizer looks at the ASCII values of the characters in the strings, going from left to right. The alphabetical order of the strings is determined by the first set of corresponding characters in the strings that do not match.

For example, if $B = \text{"fielder"}$ and $A = \text{"fieldmouse"}$, then $B < A$ is true. The two strings are the same up to the fifth character. Since the ASCII equivalent of the sixth character in string B (the character "e") is ASCII 101, and the ASCII equivalent of the sixth character in string A (the letter "m") is ASCII 109, string B is less than A .

Here are a few useful things to note when comparing strings:

- The ASCII characters follow alphabetical order from A to Z, so you can use the comparison operators to alphabetize strings.
- Uppercase characters have smaller ASCII equivalents than lowercase characters (therefore, "A" is less than "a" but "Z" is still less than "a").
- Beware of leading and trailing spaces. They can be significant when comparing strings (especially when comparing fixed length strings).

For example, " Ed" is less than "Ed", since the ASCII equivalent of the space character is 32, while "E" has an ASCII value of 69.

Retrieving Parts of a String

Retrieving part of a string is a very common BASIC operation. CA-Realizer BASIC is very flexible in how it permits you to do this. It features a number of convenient functions that allow you to easily extract a substring from anywhere in a known string. In addition, CA-Realizer provides several powerful functions that allow you to extract a substring even when the contents of the string are unknown.

Retrieving from the Left, Right, or Middle of a String

There are three very useful commands for examining and retrieving parts of strings: `Left$`, `Mid$`, and `Right$`. `Left$` returns the leftmost characters in a string, `Mid$` returns characters from the middle, and `Right$` returns the rightmost characters.

Their general form is:

```
NewString = Left$(String, NumChars)
```

```
NewString = Mid$(String, Start  
[, NumChars])
```

```
NewString = Right$(String, NumChars)
```

String is the string to process; *NumChars* is the number of characters to take from *String*.

Mid\$

For the Mid\$ function, *Start* indicates the character within the string from which to begin extracting. *NumChars* indicates the number of characters you want to retrieve from the middle of the string beginning with *Start*. If *NumChars* is not specified, all characters up to the end of the string are returned.

For example, the following statements retrieve the characters from *Salesman* ("Willie Lohman") beginning with the sixth character for a length of three:

```
Salesman = "Willie Lohman"
Middle = Mid$(Salesman, 6, 3)
PRINT Middle
e L
```

Right\$ and Left\$

Both Right\$ and Left\$ take two parameters, *String* and *NumChars*. The first denotes the string to use and the second refers to the number of characters (rightmost or leftmost, respectively) to extract.

For example, the following statements return the rightmost six characters and leftmost six characters from *Salesman*:

```
Salesman = "Willie Lohman"
LastName = Right$(Salesman, 6)
FirstName = Left$(Salesman, 6)
PRINT LastName, Firstname
Lohman, Willie
```

Plus (+) Operator

Now you can build a new string using the plus (+) operator and the pieces you just created, replacing the contents of *Salesman* with the new data as follows:

```
Salesman = LastName + ", " + FirstName
PRINT Salesman
Lohman, Willie
```

Retrieving from Any Position in the String

The examples so far have assumed that you know the contents of a string beforehand. However, this is not always the case. For instance, suppose the *Salesman* string was "Clyde Drake". The function call, `Right$(Salesman, 4)`, would return "rake," only a portion of this salesman's last name.

A more general approach to the problem is required to find and extract the last word from any string. You can do this with two functions that tell you more about the contents of a string—`Instr` and `StrTok`.

Searching with the Instr Command

`Instr`

`Instr` searches through a string for the occurrence of another string, and has the general form:

```
Location = Instr(String, WhatToFind [, Start])
```

`Instr` is particularly useful for finding keywords and single characters within a block of text. *String* is the string to search and *WhatToFind* is the string to be found.

The value returned is the location within the string where the first occurrence of *WhatToFind* was found. If *WhatToFind* does not occur within *String*, the return value will be 0.

Start, which is optional, indicates the character from which to start searching. By varying *Start*, multiple occurrences of the same string may be found.

As an example, let us say a variable named *President* has two words in it, a first name and a last name (for example, "William Taft") and you want to extract the last name.

Len

First use the `Len` function to determine the length of the string. For example, the following returns the number of characters in *President*:

```
President = "William Taft"
Size = Len(President)
PRINT Size
12
```

Now that the number of characters is known, you need to determine where the second word starts. To do this, search for the space between the first and second word using `InStr`. For example, the following searches the string *President* for a space character:

```
WordStart = InStr(President, " ")
```

`InStr` returns 8, indicating the location of the space character. This means that the second word begins at the position just after the space found by `InStr` (character 9).

You can now determine the length of the second word:

```
WordLength = Size - WordStart
PRINT WordLength
4
```

Right\$

Now use the value of *WordLength* with `Right$` to extract the last name:

```
PresName = Right$(President, WordLength)
PRINT PresName
Taft
```

In the previous example, the variables *Size*, *WordStart*, and *WordLength* are temporary. It is possible, however, to use the same string functions to produce the same result without these variables.

For example, the following statements also return the second name in the variable *President*:

```
President = "William Taft"
PRINT Right$(President, Len(President) - \
    InStr(President, " "))
Taft
```

Mid\$

You can make the above line even shorter by using the Mid\$ function:

```
President = "William Taft"  
PRINT Mid$(President, Instr(President, " ") + 1)  
Taft
```

Searching with the StrTok Command

Unlike Instr, which returns a substring location, StrTok creates a vector (one-dimensional array) where each element consists of string characters separated by a delimiter. The general form of StrTok is:

```
Array = StrTok(String [, Delimiters  
[, TokenList]][; _MarkDelims])
```

StrTok tokenizes *String* by breaking it up into elements separated by any of the characters in the string *Delimiters*, which defaults to the set of tab (ASCII 9), space (ASCII 32), new line (ASCII 10), carriage return (ASCII 13), and null (ASCII 0).

For example:

```
President = "Abraham Lincoln"  
PresName = StrTok(President)  
PRINT PresName  
Abraham  
Lincoln
```

If you look at the variable list in the debugger, you will see that *PresName*[1] contains "Abraham" and *PresName*[2] contains "Lincoln".

If you specify *TokenList*, it represents an array of strings to be considered as elements even if there are no delimiters surrounding them.

Compare the following output:

```
s = "Hello, I must be going!" : tokens = {"", ",", "!"}
PRINT StrTok(s)
PRINT StrTok(s, _Default, tokens)
Hello,
I
must
be
going!

Hello
,
I
must
be
going
!
```

If you specify the *_MarkDelims* modifier, the positions in which a string of one or more delimiters is found is indicated by an empty string in the resulting string vector. This can be useful if the inclusion of delimiters is important, or if you want "<", ">", and "<>" to be considered tokens. "<" will be found and so will ">", so you can check if there was a delimiter between them.

Replacing Text Within Strings

CA-Realizer provides the SubStr\$ function for replacing strings within other strings. Its general form is:

```
NewString = SubStr$(SourceString, Find,  
                    Replace [, Start [, NumChars]])
```

SubStr\$ is similar to InStr, except that it replaces occurrences of a given string with another string. *SourceString* is searched and all occurrences of *Find* are replaced with *Replace*. If you specify *Start*, the search begins with the *Start*th character.

If you specify *NumChars*, the search begins with *Start* and continues up to *NumChars* characters. If *Find* is an empty string, it replaces the *NumChars* characters starting at *Start* with *Replace* (in this case, if *NumChars* is not specified, *NumChars* defaults to 0). If *Find* and *Replace* are both empty, the *NumChars* characters starting at *Start* are deleted.

For example:

```
TestString = "My blue book and blue pencil."  
PRINT InStr(TestString, "blue")  
PRINT SubStr$(TestString, "blue", "red")  
PRINT SubStr$(TestString, "blue", "red", 17)  
4.0000  
My red book and red pencil.  
My blue book and red pencil.
```

Here is an example of two useful functions which can be used to remove unwanted characters from a string. The first statement defines an *exclusion* array that lists four characters to remove. Then, the *RemoveChars* function is passed the exclusion array and a string to clean. The function *CleanString* determines how many elements there are in the exclusion array using the *EndValid* function, and then loops through a call to the function that actually passes the string and removes the characters.

```
Exclude = {"(", ")", "'", ",", ""}  
PROC RemoveChars(ExcludeList, StringToClean)  
    LOCAL i  
    FOR i = 1 TO EndValid(ExcludeList)  
        CleanString(StringToClean, ExcludeList[i])  
    NEXT  
END PROC
```

```

PROC CleanString(s, ExcludeChar)
  LOCAL Pos
  Pos = Instr(s, ExcludeChar)
  WHILE Pos
    s = SubStr$(s, ExcludeChar, "")
    Pos = Instr(s, ExcludeChar, Pos)
  END WHILE
END PROC

```

The *CleanString* function removes one undesirable character at a time. It begins by using the *InStr* function to search *s* for the first occurrence of the unwanted character. It then uses the *SubStr\$* function to substitute nothing for the undesirable character (thus trimming the string) and then locates the position of the next occurrence of the unwanted character with *InStr*. *CleanString* loops until all specified characters have been removed from *s*.

The following example uses the *RemoveChars* procedure to remove the parentheses, commas, and single quotes from *DirtyString*:

```

PROC RemoveChars(ExcludeList, StringToClean)
  LOCAL i
  FOR i = 1 TO EndValid(ExcludeList)
    CleanString(StringToClean, ExcludeList[i])
  NEXT
END PROC

Exclude = {"(", ")", "'", ",", ""}
DirtyString = "T,his' '( is ))d((i)),,','r','t(y))"
RemoveChars(Exclude, DirtyString)
PRINT DirtyString
This is dirty

```

Note: There is often more than one way to manipulate strings. To obtain the best performance, use the method with the least number of functions, operators, and temporary variables.

Changing Case

When strings are compared to each other using the comparison operators, InStr and SubStr\$, or the StatSort command, uppercase and lowercase letters are considered different.

For example, the following line only matches the first instance of "Early" and returns "Late to bed and early to rise":

```
PRINT SubStr$("Early to bed and early to rise", \\  
            "Early", "Late")
```

To force uppercase and lowercase letters to be treated the same, change the _CaseSensitive system variable with the SetSys command as follows:

```
SetSys(_CaseSensitive, 0)
```

Alternatively, you can also use the LCase\$ and UCase\$ functions to explicitly convert the letters in a string to lowercase or uppercase, respectively:

```
NewString = LCase$(String)  
NewString = UCase$(String)
```

For example:

```
s = "This is A MiXed case String"  
PRINT LCase$(s)  
PRINT UCase$(s)  
this is a mixed case string  
THIS IS A MIXED CASE STRING
```

Inserting and Deleting Portions of a String

LTrim\$ and RTrim\$

The LTrim\$ and RTrim\$ functions return a copy of a string with spaces and tabs removed from the left and right of the string, respectively. Their general form is:

```
NewString = LTrim$(String)
```

```
NewString = RTrim$(String)
```

For example, LTrim\$ is used to remove the five spaces that precede the string " It's Friday":

```
PRINT LTrim$("      It's Friday")
It's Friday
```

Substrings can be inserted or deleted with StrInsert\$ and StrDelete\$.

StrInsert\$

StrInsert\$ returns a copy of *String* with the string *WhatToInsert* inserted at position *Position*.

```
NewString = StrInsert$(String, Position,
                        WhatToInsert)
```

For example:

```
PRINT StrInsert$("This house is mine", 12, "not ")
This house not is mine
```

StrDelete\$

StrDelete\$ returns a copy of *String* with *NumChars* characters removed beginning at position *Position*.

```
NewString = StrDelete$(String, Position,
                        NumChars)
```

For example:

```
s = "This is my string"
PRINT StrDelete$(s, 9, 3)
This is string
```

Converting Strings

There are times when a numeric value is stored in a string. These strings cannot be used as numeric values unless they are explicitly converted to a numeric data type. CA-Realizer BASIC provides many functions for converting strings to numbers and numbers to strings.

Numbers to Strings

NumToStr

The NumToStr function returns a string representation of a value using its default format. The general form is:

```
String = NumToStr(Value)
```

Value is some real number or a numeric variable containing a real number. For example:

```
A = 10
B = 20
PRINT A + B, NumToStr(A) + NumToStr(B)
30 1020
```

Bin\$, Oct\$, Hex\$

Note that CA-Realizer does not limit you to just decimal representations. The Bin\$, Oct\$, and Hex\$ functions convert a numeric value into a string representing the value in binary (base 2), octal (base 8), or hexadecimal (base 16), respectively.

```
String = Bin$(Value)
String = Oct$(Value)
String = Hex$(Value)
```

Note that *Value* is treated as a 32-bit unsigned integer.

Several examples are presented below:

```
n = 1234
PRINT Bin$(n)
PRINT Oct$(n)
PRINT Hex$(n)
10011010010
2322
4D2
```

Strings to Numbers and Dates

StrToNum

Just as NumToStr converts a number to a string, StrToNum converts a string to a number:

```
Value = StrToNum(String [, DefaultValue])
```

For example:

```
PRINT StrToNum("-41.000")
-41.0000
```

The value returned for a string that cannot be converted is *DefaultValue*, if specified, or 0 if *DefaultValue* is not specified.

For example:

```
Alexander = (30 + 5)
PRINT StrToNum("Alexander", 5)
5
```

Note: Val is a synonym for StrToNum and is provided by CA-Realizer for compatibility with older versions of BASIC.

StrToDate

StrToDate converts a string to a date-time value:

```
DateTime = StrToDate(String)
```

The string can contain a date and/or time value in any of a number of common formats. If the string cannot be converted, the result is 1/1/1900 0:00:00.

For example:

```
DDay = StrToDate("6-Jun-1945")
Birthday = StrToDate("12:05 AM 7/25/68")
PRINT DDay
PRINT Birthday
06/06/45
07/25/68
```

Refer to StrToDate in the *CA-Realizer Language Reference Guide* for a complete listing of accepted formats.

Just as you are not confined to decimal representations when converting numbers to strings, CA-Realizer provides a number of functions for converting from string representations of different bases to numerics.

BinToNum,
OctToNum,
HexToNum

The BinToNum, OctToNum, and HexToNum functions convert a string representing a value in binary, octal, or hexadecimal (respectively) into a decimal, unsigned integer value. The value must be between 0 and $2^{32}-1$. Their general form is:

```
Value = BinToNum(String)
```

```
Value = OctToNum(String)
```

```
Value = HexToNum(String)
```

Note that:

- Binary strings can only contain the characters 0 or 1.
- Octal strings must be comprised of the characters 0 to 7.
- Hexadecimal strings can contain the digits 0 to 9, as well as the letters A to F. CA-Realizer ignores the case of alphabetic digits in hexadecimal strings.

Several examples are presented below:

```
PRINT BinToNum("11010")
PRINT OctToNum("723")
PRINT HexToNum("A6df")
26
467
42719
```


Converting from External Formats

When manipulating data files or string buffers used with external procedures and functions, it is sometimes useful to manipulate strings representing the bytes of the binary equivalent of a value.

CA-Realizer provides you with several variants of the MKx\$ and CVx functions:

- MKI\$ and CVI operate on 2-byte integers.
- MKL\$ and CVL operate on 4-byte integers (LONGs).
- MKS\$ and CVS operate on 4-byte single precision floating point values.
- MKD\$ and CVD operate on 8-byte double precision floating point values.

The MKx\$ functions convert a value into a string, and the CVx functions convert that string back into a numeric value. Their general form is:

```
String = MKI$(Value)
String = MKL$(Value)
String = MKS$(Value)
String = MKD$(Value)

Value = CVI(String)
Value = CVL(String)
Value = CVS(String)
Value = CVD(String)
```

For example, the integer value 19520 has a 2-byte representation of 64 and 76, so MKI\$(19520) yields the 2-character string "@L" (since @ is ASCII 64 and L is ASCII 76):

```
MKI$(19520)
@L
```

The inverse function, CVI, returns 19520:

```
CVI("@L")
19520
```

Note: The MKSMBF\$ and MKDMBF\$ functions are similar to the MKS\$ and MKD\$ function, except they return their values as representations of single and double precision floating point values in the Microsoft Binary Format used by QuickBASIC. The inverse of these functions are CVSMBF and CVDMBF, respectively.

Formatting Strings

CA-Realizer BASIC provides a great amount of flexibility and control over how strings can be formatted, and provides a very powerful function named Sprint for this purpose.

The Sprint Function

The general form for Sprint is:

```
String = Sprint(FormatCode, Data1 [..., DataN])
```

FormatCode can be any formatting string that would be valid as the *FormatCode* string in the PRINT command. *Data1* through *DataN* are values to substitute into the formatting string. These values can also be arrays.

The following example formats and prints today's date using the format code D(M3 D2, Y2):

```
Today = Sprint("Today is D(M3 D2, Y2)", QDate)
PRINT Today
Today is Apr 8, 1993
```

Format Codes

Sprintf requires a format string and a list of values to format. For instance, you can produce formatted sales report data with the following string array:

```
fmt = "Item: & Quantity: ## Price: $####.##"
Name = "Widget"
Quantity = 20
Price = 295.95
PRINT Sprint(fmt, Name, Quantity, Price)
Item: Widget Quantity: 20 Price: $295.95
```

The contents of each value is inserted into the format string at the locations indicated by the format codes.

The `&` character is an example of a string format code. The following table lists valid format codes for strings. The examples refer to the string `"yam fries"`:

Format	Description	Example
<code>&</code>	Includes the complete string.	yam fries
<code>!</code>	Uses the first character from the string.	y
<code>\...\ /</code>	Prints the first n+2 characters from the string, where n is the number of characters between slashes.	yam f
<code>_ &</code>	Prints the character after the underscore.	&

There are additional formats for formatting reals, date-time values, and fractions. Refer to the PRINT command in the *CA-Realizer Language Reference Guide* for more information.

Chapter 5

Controlling the Flow of a Program

In This Chapter	5-1
Forming Statements	5-1
Control Structures	5-2
Conditional Statements	5-4
The IF Statement	5-4
The ELSE Clause	5-5
The ELSEIF Clause	5-6
The IF Function	5-7
The SELECT Statement	5-8
Exiting IF and SELECT Statements	5-9
Unconditionally Changing Flow with GOTO	5-9
Iterative Statements	5-10
The FOR...NEXT Statement	5-11
The LOOP Statement	5-13
The WHILE Statement	5-13
Exiting an Iterative Statement	5-14
Optimizing Iteration by Using Arrays	5-15
Nested Control Structures	5-17
Trapping Errors with ON ERROR GOTO	5-18
The ON ERROR GOTO Statement	5-18
Writing a Simple Error Handler	5-19
Getting Information Inside an Error-Trapping Routine	5-19
Continuing After an Error	5-21
Disabling Error-Trapping	5-21

Chapter 5

Controlling the Flow of a Program

In This Chapter

After an introduction to the basics of forming statements in CA-Realizer, you will learn how to use the various control structure statements to control the flow of your programs.

Normally, command statements execute one after the other; the general flow of a CA-Realizer program, therefore, proceeds from top to bottom.

Control statements, however, allow you to alter the ordinary top-to-bottom program flow by enabling the program to react to data values and user input. For example, FOR...NEXT and WHILE are looping control structures that allow you to execute a piece of code more than once, and IF and SELECT are decision-making control structures that allow you to execute different pieces of code based on a condition.

Forming Statements

A CA-Realizer program consists of a series of statements that modify and examine variables and constants via calls to CA-Realizer commands and functions.

Each line of a program can contain a single statement as follows,

```
PRINT "Hello"
```

or multiple statements that are separated by colons (:):

```
FOR i = 1 TO 10 : PRINT i : NEXT i
```

You can also break up statements and place them on multiple lines by ending lines with two backslashes (\\):

```
PRINT "In order to make this very long line more" \\  
      + " readable it will be broken up"
```

Programs can also contain *comments*. This allows you to put non-executing statements in your programs. A comment can be placed at the end of a program line or on its own line using either a single quote (') or the REM keyword. For example:

```
PRINT "Hello" 'This is a valid comment  
REM This is a valid comment
```

It is recommended that you use comments to explain different parts of your program, making your program logic easier to follow.

Note: In CA-Realizer, program statements do not contain line numbers and can be separated by empty lines.

Control Structures

Using control structures in your programs allows you to manage how your program operates—that is, when and how it should react to various conditions, such as data values, user input, and errors.

By default, the general flow of a CA-Realizer program proceeds from top to bottom, executing command statements one after the other. This is demonstrated by the following example. First, execute the following statements:

```
Balance = 15000  
InterestRate = .07  
Deposit = 10000  
Balance = Balance + Deposit  
Balance = Balance + (Balance * InterestRate)  
PRINT Balance
```


Now rearrange them in the following order and execute them again:

```
Balance = 15000
InterestRate = .07
Deposit = 10000
Balance = Balance + (Balance * InterestRate)
Balance = Balance + Deposit
PRINT Balance
```

Control structures allow you to break this flow—for example, you can instruct CA-Realizer to skip back and repeat a sequence of statements or choose a sequence of statements from two or more options.

To see how control structures allow you to do this, modify the previous program as follows so that its output depends upon the value of its variables:

```
Balance = 15000
IF Balance > 50000 THEN
    InterestRate = .07
ELSE
    InterestRate = .05
END IF

Deposit = 10000
IF Deposit >= 5000 THEN
    PRINT "Toaster Oven"
ELSE
    PRINT "Key Chain"
END IF

Balance = Balance + (Balance * InterestRate)
Balance = Balance + Deposit
PRINT Balance
```

The value of the *InterestRate* variable depends upon the value of the *Balance* variable. Since *Balance* is not greater than 50,000, the interest rate is set at 5%. If you change the value of *Balance* to 115,000, then the interest rate becomes 7%.

By running the previous program with different values for *Balance* and *Deposit*, you can see how the value of data determines the outcome. You can divert the flow of this program along four possible paths, determined by the *Balance* and *Deposit* variables. Only command statements along the chosen path are executed.

In general, control statements fall into two categories: *conditional* and *iterative*. Conditional statements control program flow according to the occurrence of some state or condition. Iterative statements execute a block of program statements one or more times.

Conditional Statements

There are three conditional statements in the CA-Realizer language—IF, SELECT, and ON ERROR GOTO. With these statements, a CA-Realizer program can choose an action based on one or more conditions. A discussion of the IF and SELECT statements follows; ON ERROR GOTO is discussed later in this chapter.

The IF Statement

So far you have seen a few examples of the IF statement in action. The IF statement is one of the most fundamental control structures, and it can be used in a number of ways.

The format for a simple IF statement is as follows:

```
IF Condition THEN  
    Statements  
END IF
```

Condition is an expression that returns a true or false value. The expression can consist of functions and conditional operators.

For example:

```
IF Name = "Manager" THEN  
    Price = Price * Discount  
END IF  
  
IF Population > 1.0E+6 THEN  
    PRINT "Overpopulated"  
    Transport = Population * Ratio  
END IF
```

You can use the result of a computation as an IF statement condition. If the result is non-zero, the condition is true; if the result is 0, the condition is false. For example:

```
IF InStr(Name, "Robert") THEN
    PRINT "Hello Bob"
END IF
IF ItemTotal THEN
    PRINT "Number of items in stock: ", ItemTotal
END IF
```

The ELSE Clause

You can also supply an IF statement with program statements that execute when a condition fails. Use the ELSE keyword to extend the format of the IF statement as follows:

```
IF Condition THEN
    Statements1
ELSE
    Statements2
END IF
```

The program statements in the ELSE portion of an IF statement are executed if the condition fails. For example:

```
IF (Math > 750) AND (Verbal > 700) THEN
    PRINT "Accepted"
ELSE
    PRINT "Rejected"
END IF
```

By nesting IF statements within IF statements, you can perform actions based on more than one condition. For instance, the IF statements below classify a value based upon the value range in which it falls:

```
Grade = 89
IF Grade < 60 THEN
    PRINT "F"
ELSE
    IF Grade < 70 THEN
        PRINT "D"
    ELSE
        IF Grade < 80 THEN
            PRINT "C"
        ELSE
            IF Grade < 90 THEN
                PRINT "B"
            ELSE
                PRINT "A"
            END IF
        END IF
    END IF
END IF
```

The ELSEIF Clause

You can extend the format of an IF statement even further by adding the ELSEIF keyword. The ELSEIF construct, shown below, allows you to create an IF statement with more than two possible outcomes:

```
IF Condition THEN
    Statements1
ELSEIF
    Statements2
    .
    .
    .
ELSEIF
    StatementsN
ELSE
    StatementsN + 1
END IF
```

Therefore, you can use ELSEIF to avoid the nested IF statements in the previous example by rewriting it as follows:

```
IF Grade < 60 THEN
    PRINT "F"
ELSEIF Grade < 70 THEN
    PRINT "D"
ELSEIF Grade < 80 THEN
    PRINT "C"
ELSEIF Grade < 90 THEN
    PRINT "B"
ELSE
    PRINT "A"
END IF
```

The IF Function

You can also use the IF statement as a function, conditionally setting the value of a scalar or an array. The format is:

```
ReturnValue = IF Condition THEN Data1 ELSE Data2
```

The following example sets the value of a string scalar:

```
Final = IF Grade >= 60 THEN "Pass" ELSE "Fail"
```

The content of the *Final* string depends on the value of *Grade*.

If any part of an IF function contains an array, it results in an array. For example, if *DealerPrice* and *NormalPrice* are arrays, then the statement below creates an array called *FinalPrice*:

```
FinalPrice = IF Person = "Dealer" THEN \\  
    DealerPrice ELSE NormalPrice
```

The SELECT Statement

IF...THEN...ELSE statements can quickly become difficult to follow when there are many different conditions to check. In such cases, it is better to use the SELECT statement as it more clearly and efficiently handles these situations.

SELECT evaluates an expression and compares its value to a number of values (or ranges). When the proper range is found, CA-Realizer executes the block of program statements associated with it.

The format of the SELECT statement is as follows:

```
SELECT CASE Expression
        CASE Range1
            Statements1
        .
        .
        .
        CASE RangeN
            StatementsN
        [CASE ELSE
            StatementsN + 1]
END SELECT
```

Each CASE of a SELECT statement contains a defined range and a corresponding set of program statements. Program statements following the CASE ELSE are executed if CA-Realizer does not find a matching range.

There are several ways to specify a range. You can define a range as a single value or multiple values. For example:

```
CASE 3
CASE 4, 5
CASE "Empty"
CASE "Full", "Loaded"
```

You can also define ranges to be all the values between two extremes or all the values satisfying a relation. For instance, to define a range as the numbers from 10 to 20 and all numbers greater than 50, use the following:

```
CASE 10 TO 20, IS > 50
```

The next line matches all strings which fall alphabetically between "horse" and "iguana":

```
CASE "horse" TO "iguana"
```

You can also use the IS keyword with any comparison operator:

```
CASE IS < 18, IS >= 26
```

Now rewrite the grading program described previously, replacing the IF statement with a SELECT statement:

```
SELECT CASE Grade
  CASE IS < 60
    PRINT "F"
  CASE 60 TO 69
    PRINT "D"
  CASE 70 TO 79
    PRINT "C"
  CASE 80 TO 89
    PRINT "B"
  CASE ELSE
    PRINT "A"
END SELECT
```

Exiting IF and SELECT Statements

You can interrupt the IF and SELECT statements using EXIT commands. When an EXIT IF is encountered, execution immediately jumps to the first program statement after the END IF statement. When an EXIT SELECT is encountered, execution jumps to the program statement after the END SELECT statement.

Unconditionally Changing Flow with GOTO

CA-Realizer provides one unconditional statement, GOTO. It allows the program flow to jump unconditionally from one location to another; its prototype is as follows:

```
GOTO Label
.
.
.
Label:
.
.
```

The target of a GOTO is a label. Labels can be any legal identifier, followed by a colon, that is on a line by itself.

Use caution when using GOTOs. If the program attempts to jump to the middle of a FOR loop, the loop variable and end condition will not be initialized, and the results will be unpredictable. Additionally, it is illegal to use GOTO to go into a procedure or function that ends, or out of a procedure or function, without returning and allowing that procedure or function to terminate normally. In other words, you cannot use GOTO to undo the sequence of function calls.

Note: While not encouraged in structured programming, GOTO does have its uses, particularly for trapping errors and exiting a number of nested loops of the same type. In most cases, however, the GOTO statement can be replaced with a WHILE or FOR loop.

Iterative Statements

An *iterative statement* executes a block of program statements one or more times. There are three kinds of iterative statements supported by CA-Realizer—FOR...NEXT, LOOP, and WHILE.

While there is some overlap, you would typically use these statements under different situations. Here are some guidelines:

- Use the FOR...NEXT statement when you want to execute one or more statements a predetermined number of times.
- Use LOOP...END LOOP when you do not initially know how often you intend to execute a group of statements.
- Use the WHILE...END WHILE statement when you want to execute a block of statements until some condition is met or changes.

The FOR...NEXT Statement

The FOR...NEXT statement is probably the single most common iterative statement. This statement loops through a block of program statements while it counts from a beginning value to an ending value.

It has the following format:

```
FOR LoopVariable = Begin TO End
    Statements
NEXT LoopVariable
```

The loop variable contains the current value of the loop count. It starts with the beginning value (*Begin*) and is incremented by a value of 1 each time the loop repeats. CA-Realizer exits the loop when the loop count exceeds the ending value (*End*).

The loop variable can also be used by program statements within the loop. For example, the following FOR loop prints the numbers from 1 to 10:

```
FOR k = 1 TO 10
    PRINT k
NEXT k
PRINT USING "The final value of k is ##"; k
```

Note: 11 is the final value of *k* in this example, which is where the loop terminates.

To increment the loop counter by a value other than 1 on each iteration, add the STEP option to the FOR...NEXT statement:

```
FOR LoopVariable = Begin TO End STEP Increment
    Statements
NEXT LoopVariable
```

The increment can be any real number, a fraction, or even a negative number. The increment value is added to the loop variable after every iteration. For example, the following loop counts by three from 1 to 20:

```
FOR k = 1 TO 20 STEP 3
    PRINT k
NEXT k
PRINT USING "The final value of k is ##"; k
The final value of k is 22
```

A FOR statement with a negative increment counts backwards. The beginning value should be greater than the ending value; CA-Realizer exits the loop when the loop count is less than the ending value. The following example counts down from 10 to 1:

```
FOR k = 10 TO 1 STEP -1
    PRINT k
NEXT k
PRINT "Blast Off!"
```

The LOOP Statement

The LOOP statement is a simple iterative statement—it repeats a set of program statements between the LOOP and END LOOP until an EXIT LOOP is encountered. If no EXIT LOOP command is reached, the program loops forever.

```
LOOP
    Statements
END LOOP
```

The following LOOP statement iterates until the value of *k* is greater than 10:

```
k = 1
LOOP
    PRINT k
    k = k + 1
    IF k > 10 THEN
        EXIT LOOP
    END IF
END LOOP
PRINT USING "The final value of k is ##"; k
```

As with the FOR...NEXT example above, 11 is the final value of *k*.

The WHILE Statement

The WHILE statement evaluates a condition before each iteration. If the condition is true, it executes the program statements within the loop. The loop continues until the condition is false.

```
WHILE Condition
    Statements
END WHILE
```

You can rewrite the previous LOOP example with a WHILE statement. This removes the IF statement from the body of the loop, as shown below:

```
k = 1
WHILE k <= 10
    PRINT k
    k = k + 1
END WHILE
PRINT USING "The final value of k is ##"; k
```

Exiting an Iterative Statement

Like the conditional statements, you can exit all three of the above iterative statements using EXIT commands. For instance, the EXIT LOOP command enables you to terminate a LOOP statement. When an EXIT LOOP is encountered, CA-Realizer skips to the program statement after END LOOP.

The EXIT FOR and EXIT WHILE commands work similarly. When either of these EXIT commands is reached, the loop is terminated.

Note: You can use the WEND keyword in place of END WHILE.

An EXIT command always corresponds to the innermost control structure of its type. For instance, the EXIT WHILE statement in the following example causes CA-Realizer to skip to the program statement after the END WHILE. The IF statement, the FOR loop, and the WHILE loop are all terminated:

```
k = 1
WHILE k <= 10
  FOR j = 1 TO 10
    PRINT k * j
    IF k * j > 75 THEN
      EXIT WHILE
      PRINT "Never Executed"
    END IF
  NEXT j
  k = k + 1
END WHILE
PRINT USING "The final value of k is ##"; k
```

Optimizing Iteration by Using Arrays

CA-Realizer's array operations are much faster than executing statements in a loop. Therefore, whenever possible, use an array operation instead of a loop to manipulate any CA-Realizer data types.

You can save a large amount of execution time with array operations, particularly for manipulating large data structures. For example, to add two 3000 element arrays, you might use the following loop:

```
FOR k = 1 TO 3000
    Total[k] = FirstScore[k] + SecondScore[k]
NEXT k
```

The array operation below, however, is much faster and easier to write:

```
Total = FirstScore + SecondScore
```

Here is an example of how you can convert loops to array operations. The following sample program fragment loops through a point-of-sales database and computes the total number of sales for a particular salesperson. The salesperson is identified by a number stored in the *PersonID* variable:

```
TotalSales = 0
FOR k = 1 TO EndValid(Sales.Item)
    IF Sales.Seller[k] = PersonID THEN
        TotalSales = TotalSales + Sales.Amount[k]
    END IF
NEXT k
PRINT TotalSales
```

You can remove the FOR statement—which sums *sales*—by computing *TotalSales* with an array operation instead:

```
PersonAmounts = IF Sales.Seller = PersonID \\  
                THEN Sales.Amount ELSE 0  
TotalSales = Sum(PersonAmounts)  
PRINT TotalSales
```

The IF function creates an array called *PersonAmounts*. The elements in *PersonAmounts* correspond directly to the elements in the point-of-sales database. Only sales made by the salesperson identified with *PersonID* are placed in the *PersonAmounts* array. Elements of *PersonAmounts* corresponding to other salespeople are set to 0. The Sum function returns the total number of sales in the *PersonAmounts* array.

Here is another way of implementing the previous example by using a comparison operation instead of the IF function:

```
Matches = (Sales.Seller = PersonID)
PersonAmounts = Matches * Sales.Amount
TotalSales = Sum(PersonAmounts)
PRINT TotalSales
```

Matches is a Boolean array (an array of 1's and 0's) created by the comparison operation. Elements of the *Matches* array contain a 1 when the comparison operation is true and 0 when the comparison is false. Multiplying *Matches* by the *Sales.Amount* array produces the *PersonAmounts* array.

Although these array-handling techniques are typically more efficient, sometimes iterative statements are the only way to maintain full control over the computation of a solution. You may come across this situation occasionally when manipulating individual dimensions in a multidimensional array or when manipulating elements of some other complex structure.

Nested Control Structures

You can nest control structures within each other. For example, a LOOP structure can contain an IF statement, and a FOR loop can contain a WHILE statement or another FOR loop. Control structures can be nested up to approximately 128 levels deep.

If CA-Realizer finds an EXIT statement, it corresponds to the innermost loop of that type, as illustrated in the following example:

```
'A customer walks up to the counter of the hardware
'store with NumberItems items. This program fragment
'computes the total price of those items,
'considering a discount and a few other conditions

WHILE i < NumberItems
  IF Price[i] > 10.00 THEN
    'Discount items over $10.00
    SELECT CASE ProductName[i]
      CASE "Hammer"
        Discount = 0.07
      CASE "Screwdriver"
        Discount = 0.09
      CASE "Garden Hose"
        'No longer carried. Leave the IF
        'statement - do not add it to the sale.
        EXIT IF
      CASE "Dynamite"
        IF Age < 21 THEN
          'Too young to buy explosives
          'Stop the entire sale
          EXIT WHILE
        END IF
    END SELECT
    NewPrice = Price[i] * (1 - Discount)
    Total = Total + NewPrice
    PRINT "New price is ", NewPrice
  ELSE
    Total = Total + Price[i]
  END IF

  'Continue here after an EXIT IF. The WHILE loop
  'still runs.
  i = i + 1
END WHILE

'Continue here after an EXIT WHILE
```

The EXIT IF statement breaks out of the IF statement to prevent an out-of-stock item from being added into the sale. The EXIT WHILE breaks out of the entire WHILE loop to prevent any more of the sale from being computed.

Trapping Errors with ON ERROR GOTO

The ON ERROR GOTO statement allows you to trap errors before they force your program to halt. Without some form of error trapping, runtime errors (for example, trying to open a non-existent file) cause CA-Realizer to display an error message and stop.

The ON ERROR GOTO Statement

ON ERROR GOTO allows your program to branch to an error-handling routine that you have written. Typically, these routines correct the error and resume execution of the application.

Using the following general syntax, you can enable error trapping in a certain part of your program:

```
ON ERROR GOTO Label      'Set error trap
```

This statement instructs the program to go to the line identified by *Label* when it encounters an error.

The label and the error-handling routine can be placed in any module in your program, provided that the module containing the label has been called **before** the error condition occurred.

You must make sure that the label is not encountered as part of normal procedural processing. A good way of doing this is to use an EXIT MACRO or EXIT PROGRAM command before the label. For example:

```
ON ERROR GOTO ErrorHandler:
.
.
.
EXIT MACRO
Error handler:
.
.
```

Writing a Simple Error Handler

An error-handling routine typically has three components:

- A label, which is referred to by the ON ERROR GOTO statement
- A procedure call or a routine which determines the cause of the error and takes appropriate action
- At least one RESUME or RESUME NEXT command

However, before you can write an error handler, you need the services of three additional functions—ERR, ERF, and ERL.

Getting Information Inside an Error-Trapping Routine

You can use the ERR, ERL, and ERF functions inside an error-trapping routine to provide information about the error. ERR returns a numeric code for the most recent runtime error, ERF returns the name of the file in which the error occurred, and ERL returns the line number at which the error occurred.

Typically, you use ERR as part of a SELECT statement. In the following example, assume that error number 25 indicates that a file could not be opened (for example, if the wrong floppy disk was in the disk drive).

```
ON ERROR GOTO ErrorHandler:
.
.
.
EXIT MACRO

ErrorHandler:
SELECT CASE ERR
CASE 25
    x = MsgBox("Unable to open "+filename, \
        "Error!", _MB_RetryCancel, \
        _MB_Exclamation)
    SELECT CASE x
        CASE _Retry
            'try again
            RESUME
        CASE _Cancel
            RESUME NEXT
    END SELECT
CASE ELSE
    'An unanticipated error has occurred. Stop
    'execution and display the error message
    ON ERROR GOTO 0
END SELECT
```

When error condition 25 is encountered, the error handler displays a dialog box to inform the user that the file cannot be opened. The program then waits for the user to choose either the **RETRY** or **CANCEL** button.

If the user chooses **RETRY**, the program executes the **RESUME** statement, causing CA-Realizer to branch back to the exact statement that caused the error (presumably after the correct disk has been mounted) and tries to open the file again. Choosing the **CANCEL** button causes processing to continue from the line following the one that caused the error condition (**RESUME NEXT**).

A complete list of error numbers can be found in the Release Notes file installed with CA-Realizer.

Continuing After an Error

Once an error occurs and the error-trapping routine has done what it can to correct the error, the error-trapping routine can then do one of the following:

- Attempt to execute the offending statement again

When used alone, `RESUME` returns control back to the command that caused the error. In the case of a true program error not caused by the user, this can result in an infinite loop.

- Skip the statement and continue execution

`RESUME NEXT` tells CA-Realizer to skip the offending statement and resume execution at the following statement. If the error statement that is skipped sets a needed value, this can cause other errors later in the program.

- Jump to another part of the program

`RESUME Label` transfers control to the first statement after *Label*.

- Report an error and stop the program execution

Including the `ON ERROR GOTO 0` statement *inside* the error-handling routine causes CA-Realizer to print the standard error message for that error condition and stops the program.

- Stop the program execution

The program execution can be stopped explicitly with the `EXIT PROGRAM` command or the `STOP` command. This can be useful if the error occurred in an asynchronous procedure called as a result of user interaction in a menu, form, or other tool. `EXIT PROGRAM` clears the error condition and allows other events to be processed.

Disabling Error-Trapping

If the `ON ERROR GOTO 0` statement appears *outside* of an error-handling routine, it switches off error handling.

Chapter 6

Structured Programming with Procedures and Functions

In This Chapter	6-1
Using Procedures and Functions	6-1
Determining When to Use a Procedure or Function	6-2
Defining a Procedure or Function	6-2
Declaring a Procedure	6-3
Declaring a Function	6-5
Using Parameters in Procedures and Functions	6-7
Formal and Actual Parameters	6-8
General Rules for Using Parameters	6-9
Mixing Formal Parameters and Variables	6-10
Passing Parameters by Reference or Value	6-10
Scoping Variables in Procedures and Functions	6-13
Global Variables	6-13
Local Variables	6-13
Using Modifiers in Procedures and Functions	6-15
Including Optional Parameters and Modifiers	6-16
Obtaining the Number of Optional Parameters and Modifiers	6-17
Referencing Optional Parameters and Modifiers	6-18
Redefining Procedures and Functions	6-20
Functions and Expression Evaluation	6-20
Recursion	6-21

Creating and Running CA-Realizer Modules	6-22
Creating Libraries	6-22
Using Modules as Procedures	6-22
Setting the Hourglass Cursor	6-23
Using the EXECUTE Command	6-23

Chapter 6

Structured Programming with Procedures and Functions

In This Chapter

This chapter describes how to incorporate procedures and functions in your CA-Realizer programs.

Using Procedures and Functions

Generally, procedures and functions are used to encapsulate frequently-used program code or complex algorithms. Procedures and functions can contain any valid program statements.

Blocks of statements that are repeated throughout a program can simply be defined once as a procedure or function, making it easier for you to manage and maintain your code. In addition, placing frequently-used code within a procedure or function allows you to easily use that code in other programs.

Structuring complex programs into smaller components of procedures and functions makes it easier to understand, develop, test, and modify programs.

Note: Procedures and functions replace the GOSUB and unstructured GOTO statements of earlier versions of BASIC.

Determining When to Use a Procedure or Function

Procedures and functions serve different purposes. As a general rule, use procedures for program statements that modify more than one variable or encapsulate a particular process, and use functions for algorithms that return either a single value, family, or array.

Defining a Procedure or Function

Like variables, procedures and functions are identified by their name. The names of procedures and functions must meet the same requirements as variable names. For example, they cannot contain spaces or start with a number. (Naming conventions are detailed in the Valid Variable Names section in the “Variables and Data Types” chapter of this guide.)

The name of a procedure or function represents the blocks of program statements defined to that procedure or function. Elsewhere in your program, you execute these blocks of code by simply referencing the procedure/function name. This is known as *calling* or *invoking* the procedure/function.

For example, if you have defined a function named *AddThese* somewhere in your application that calculates the sum of two given values, you can legally call that function with the following statement:

```
total = AddThese(4, 5)
```

When CA-Realizer encounters a reference to a procedure or function name, processing jumps to the first line of that procedure or function, and the procedure or function is executed in its entirety. When the procedure or function has completely executed, program control is sent to the line immediately following the statement that called the procedure or function.

Important! Before you can use a procedure or function, you must define (or declare) it. Instructions on declaring procedures and functions follow.

Note: Commonly-used procedures and functions should be defined at the beginning of the program or placed in a separate file.

Declaring a Procedure

Procedures in CA-Realizer are defined as follows:

```
PROC Name
    Statements
END PROC
```

where *Name* is the name of the procedure being defined, and *Statements* is the block of valid CA-Realizer code defined as the new procedure. The PROC and END PROC commands mark the beginning and end of the procedure, respectively.

Note: You can optionally use the SUB command as a synonym for PROC.

By default, when a procedure is called, all statements between PROC and END PROC are executed in their entirety. You can, however, include an EXIT PROC command to exit the procedure at a particular point. (EXIT PROC is discussed in more detail later in this section.)

Let us now create a procedure named *Count* which contains a simple counting loop. Type the following lines:

```
PROC Count
    FOR k = 1 TO 10
        PRINT k
    NEXT k
END PROC
```

Highlight the entire *Count* procedure and execute it with the RUN INSTANT command. The procedure is defined, yet nothing appears in the Print Log.

Now enter and execute the following statements to call the procedure name. This runs the program statements it contains and then prints the results:

```
Count  
PRINT "The final value of k is"; k
```

As you can see, invoking the *Count* procedure in your program instructs CA-Realizer to print the numbers 1 to 10 in the Print Log.

Now let us demonstrate the EXIT PROC command by modifying *Count* as follows:

```
PROC Count  
  FOR k = 1 TO 10  
    PRINT k  
    IF k = 5 THEN  
      EXIT PROC  
    END IF  
  NEXT k  
END PROC
```

This modified version forces CA-Realizer to exit the procedure when *k* equals five. If you run the following statements, only 1 to 5 are printed:

```
Count  
PRINT "The final value of k is"; k
```

Note: You cannot use EXIT statements within a procedure to exit from conditional or iterative statements that are started outside the procedure. For example, you cannot use EXIT IF within a procedure to leave an IF statement that starts outside the procedure. An EXIT IF statement can be used within a procedure only if the IF statement is contained within the procedure. Likewise, an END statement cannot be used in a procedure to end blocks started outside the procedure.

Declaring a Function

Functions are in many ways similar to procedures, but work in a slightly different manner. Typically, a function should contain a set of commands that yields a single result (usually a scalar, a family, or an array). When a function is finished, it passes the result back to the program statement from which it was called.

Functions in CA-Realizer are defined as follows:

```
FUNC Name
    Statements
    RETURN Expression
END FUNC
```

Name is the name of the function being defined and *Statements* are the blocks of valid CA-Realizer code defined to the new function. The RETURN statement instructs CA-Realizer to evaluate the given *Expression*, pass the result to the program statement that called the function, and then terminate the function. The FUNC and END FUNC commands mark the beginning and end of a function (respectively).

Note: You can have more than one RETURN command in a function, but CA-Realizer processes only the first one it reaches.

When a function is called, all statements between FUNC and END FUNC are executed in their entirety. Note that unlike procedures, no EXIT FUNC statement is allowed—you must use the RETURN statement.

Let us now create a function named *LetterScore* that determines the letter grade (such as A or F) that corresponds to a given numeric grade. Type the following lines:

```
FUNC LetterScore
  SELECT CASE Grade
    CASE IS < 60
      RETURN "F"
    CASE 60 TO 69
      RETURN "D"
    CASE 70 TO 79
      RETURN "C"
    CASE 80 TO 89
      RETURN "B"
    CASE ELSE
      RETURN "A"
  END SELECT
  PRINT "This statement is never executed."
END FUNC

Grade = 89
PRINT LetterScore
```

In this example, the *LetterScore* function is invoked when CA-Realizer processes the `PRINT LetterScore` statement. Since the given *Grade* is 89, the *LetterScore* function returns the letter "B" to the `PRINT` statement.

Each `CASE` of the `SELECT` statement in the example returns a string with a single letter. Note that CA-Realizer never executes the `PRINT` statement that appears before the `END FUNC` statement because functions always terminate after executing a `RETURN` command. The `CASE ELSE` statement guarantees that one of the cases will be executed.

Here is another example; it prompts the user for a last name, searches for the name in an array, and returns either the index of the first occurrence or a 0 if the name cannot be found:

```
FUNC GetPerson
  INPUT "Salesman's last name?"; SalesName
  FOR k = 1 TO EndValid(LastNames)
    IF SalesName = LastNames[k] THEN
      PersonID = k
      EXIT FOR
    END IF
  PersonID = 0
NEXT k
RETURN PersonID
END FUNC
```

The INPUT statement prompts the user for a last name and passes the name entered by the user to the *SalesName* variable. If the name exists in the *LastNames* array, it returns the index of the first occurrence. If the name is not found, the *GetPerson* function returns a zero.

Note: This function can actually be replaced with the CA-Realizer FirstMatch command.

Since function execution ends when it reaches a RETURN command, you can rewrite the *GetPerson* function to make better use of the RETURN command as follows:

```
FUNC GetPerson
    INPUT "Salesman's last name?"; SalesName
    FOR k = 1 TO EndValid(LastNames)
        IF SalesName = LastNames[k] THEN
            RETURN k
        END IF
    NEXT k
    RETURN 0
END FUNC
```

You no longer need the temporary variable, *PersonID*.

To invoke the *GetPerson* function from within a program statement, enter the following lines:

```
Person = GetPerson
IF Person THEN
    PRINT "Salesperson found: "; SalesPerson[Person]
ELSE
    INPUT "Can't find salesman", "Error";
END IF
```

Using Parameters in Procedures and Functions

You can design procedures and functions to act upon a set of values and then modify their behavior using these values. Data is passed to a procedure or function through a *parameter*.

A parameter represents a value and is given a name like a variable. (When you define multiple parameters, it is often referred to as a *parameter list*.) Use parameters like variables—for example, in expressions, built-in functions, and assignment statements.

When you define a procedure or function, you must also define its parameters using the following general format:

```
PROC Name(Parameter1 [..., ParameterN])  
    Statements  
END PROC  
  
FUNC Name(Parameter1 [..., ParameterN])  
    Statements  
END FUNC
```

Parameters must be listed in parentheses after the procedure or function name. If passing more than one parameter, parameters must be separated with commas.

For example, we previously defined a procedure named *Count* as follows:

```
PROC Count  
    FOR k = 1 TO 10  
        PRINT k  
    NEXT k  
END PROC
```

We can now modify *Count* to use a parameter as follows:

```
PROC Count(Last)  
    FOR k = 1 TO Last  
        PRINT k  
    NEXT k  
END PROC
```

Formal and Actual Parameters

When you declare a parameter within a procedure or function, it is called a *formal* parameter. You can pass a parameter to a procedure or function that is a variable, constant, or expression declared outside of the procedure or function. These passed values are called *actual* parameters.

Formal Parameters Because formal parameters are declared *within* procedures/functions, you can only reference them while the procedure or function in which they are declared is executing. When a procedure or function terminates, all of its parameter definitions are lost.

Actual Parameters As stated above, when you pass a variable, constant, or expression to a procedure or function that is declared outside of the procedure or function, the passed value is referred to as an *actual* parameter.

To demonstrate the difference between formal and actual parameters, try the *Count* procedure we modified above by supplying a parameter when you invoke it:

```
PROC Count (Last)
  FOR k = 1 TO Last
    PRINT k
  NEXT k
END PROC

UpTo = 25
Count (UpTo)
```

In this example, *UpTo* is an actual parameter and *Last* is a formal parameter. When the *Count* procedure terminates, *Last* is no longer defined (in fact, if you try to access *Last* outside the *Count* procedure, CA-Realizer reports an error in the Error Log). However, *UpTo* still retains its value.

General Rules for Using Parameters

Here are two simple rules for using actual and formal parameters:

- Each actual parameter must correspond to each formal parameter declared in the function or the procedure. (In other words, you cannot have a different number of actual parameters and formal parameters.)
- Parameters must appear in the same order in which they are defined.

For example, the following procedure—*Count2*—declares two formal parameters. When you invoke *Count2*, you must supply two actual parameters (the first is the beginning value of the FOR loop and the second is the ending value).

```
PROC Count2(First, Last)
  FOR k = First TO Last
    PRINT k
  NEXT k
END PROC
```

```
StartAt = 5
UpTo = 25
Count2(StartAt, UpTo)
```

The following two lines incorrectly invoke the *Count2* procedure because the number of actual and formal parameters differ:

```
Count2(UpTo)
Count2(UpTo, StartAt, UpTo)
```


Mixing Formal Parameters and Variables

You can mix formal parameters with variables in an expression. You can also assign a value to a parameter, or assign a parameter to a variable.

A formal parameter can have the same name as a variable that is defined outside the procedure or function. There is no conflict because the formal parameter name is visible to the program only during the execution of the procedure or function in which it is declared. When a procedure or function terminates, the formal parameter name is lost.

For example, you can create a variable named *Last* before executing the *Count2* procedure. The *Last* variable defined outside of the procedure does not interfere with the *Last* formal parameter declared within the *Count2* procedure.

```
PROC Count2(First, Last)
  FOR k = First TO Last
    PRINT k
  NEXT k
END PROC

Last = 99
StartAt = 5
UpTo = 25
Count2(StartAt, UpTo)
PRINT Last
```

Passing Parameters by Reference or Value

Procedures and functions that modify their parameters should do so with caution. CA-Realizer treats passed parameters differently depending on the *type* of the value that is passed by the parameter.

By default, variables are *passed by reference*, and expressions and constants are *passed by value*. You can, however, force a variable to be passed by value.

These different conventions are described below.

Passing By Reference When you pass a variable to a procedure or function, CA-Realizer uses the pass by reference calling convention.

This means that the formal parameter associated with the passed variable is used as an alias for the passed variable—therefore, if the value of the formal parameter is changed during the execution of the procedure/function, the value of the corresponding variable that exists outside of the procedure/function is also changed.

If, for example, a change is made to *Last* in the *Count2* procedure, the value of the variable named *UpTo* is changed as well.

```
PROC Count2(First, Last)
  FOR k = First TO Last
    PRINT k
  NEXT k
  Last = 0
END PROC

StartAt = 5
UpTo = 25
Count2(StartAt, UpTo)
PRINT UpTo
```

The *UpTo* variable is passed by reference to the *Count2* procedure. The formal parameter named *Last* references the *UpTo* variable directly. When *Count2* sets its *Last* parameter to 0, the value of *UpTo* is also changed to 0.

Passing By Value When you pass an expression or constant to a procedure or function, CA-Realizer uses the pass by value calling convention. In this case, any change made to the formal parameter that is associated with the passed expression or constant remains local to the procedure or function (modifications are *not* sent back to the passed value).

For example, change one of the values passed to the *Count2* procedure from *UpTo* to *UpTo*2* (thereby forming an expression):

```
PROC Count2(First, Last)
  FOR k = First TO Last
    PRINT k
  NEXT k
  Last = 0
END PROC
```

```
StartAt = 5
UpTo = 25
Count2(StartAt, UpTo * 2)
PRINT UpTo
```

In this case, the *UpTo* variable is passed by value to the *Count2* procedure. When *Count2* sets its *Last* parameter to 0, the value of *UpTo* remains unchanged.

Passing a Variable By Value

You can also easily force a variable to be passed by value by enclosing it in parentheses to make it an expression.

For example, simply invoke *Count2* with the *UpTo* variable as an expression:

```
StartAt = 5
UpTo = 25
Count2(StartAt, (UpTo))
```

In this case, *UpTo* is passed by value to *Count2* and is not changed when the *Last* formal parameter is set to 0.

Scoping Variables in Procedures and Functions

The term *scope* is used to refer to the *visibility* of a variable—that is, where in the program the variable is active. In CA-Realizer, variables can be either *global*—visible over the entire application or *local*—visible only in the module function or procedure that owns it and procedures and functions it calls.

By default, if a variable is not explicitly scoped, it is global.

Global Variables

All the variables used in our examples so far have been *global*. Global variables can be accessed from any part of a program. After declaring a global variable, it remains defined until you remove it with the RESET or CLEAR command.

Local Variables

Like formal parameters, local variables remain defined only during the execution of the procedure or function in which they are declared. When execution completes, local variables disappear.

Therefore, it is recommended that variables used as loop counters or temporary values in a procedure or function be declared as local. For example, in the *Count2* procedure presented in earlier sections of this chapter, the counting variable, *k*, is a global variable. When the procedure terminates, *k* is still defined.

Because the value of *k* may be adversely affected if you use the variable name *k* anywhere else in your program, it is recommended that you declare *k* as a local variable in the *Count2* procedure.

To declare a local variable, use the `LOCAL` command:

```
LOCAL Variable1 [..., VariableN]
```

Typically, local variables are declared at the beginning of the procedure, function, module, or program. Note that you can declare multiple variables with a single `LOCAL` statement—simply supply a list of variables separated by commas.

Similar to formal parameters, local variables can have the same name as a variable that is defined outside the procedure or function. There is no conflict because the local variable name is visible to the program only during the execution of the procedure or function in which it is declared. When a procedure or function terminates, the previous variable remains untouched.

For instance, to make the counting variable, *k*, local to the *Count2* procedure, do the following:

```
PROC Count2(First, Last)
  LOCAL k
  FOR k = First TO Last
    PRINT k
  NEXT k
END PROC
```

You can now call *Count2* from within the following loop:

```
FOR k = 1 TO 5
  PRINT "Begin Iteration", k
  Count2(1, 10)
  PRINT "End Iteration", k
NEXT k
```

The local variable, *k*, defined inside *Count2* does not interfere with the global value, *k*, defined outside *Count2*.

Note: Any procedure or function that a procedure or function calls, can use the local variables defined in the procedure or function.

Tip: The proper use of local variables conserves memory and prevents accidental variable conflicts.

Using Modifiers in Procedures and Functions

Procedures and functions also accept a second set of parameters called *modifiers*. Modifiers can be used to separate the information passed to a procedure or function into two distinct lists. They can affect how a parameter value is processed or supply a secondary value to an equation.

You can include a list of modifiers in a procedure or function declaration using the following formats:

```
PROC Name([Parameter1 [..., ParameterN]] \\  
          [; Mod1 [..., ModN])  
    Statements  
END PROC  
  
FUNC Name([Parameter1 [..., ParameterN]] \\  
          [; Mod1 [..., ModN])  
    Statements  
END FUNC
```

Notice that the list of modifiers must be separated from the regular parameter list with a semicolon. Also, multiple modifiers in a single list must be separated by commas.

Now, create a new procedure called *Count3*. This adds a modifier called *Increment* after the *First* and *Last* parameters that sets the increment of the counting loop:

```
PROC Count3(First, Last; Increment)  
    LOCAL k  
    FOR k = First TO Last STEP Increment  
        PRINT k  
    NEXT k  
END PROC
```

You must invoke *Count3* with two parameters and one modifier. For example:

```
Count3(StartAt, UpTo; 2)
```

Including Optional Parameters and Modifiers

You can create flexible parameter lists for procedures and functions by mixing required and optional parameters. Simply place two periods at the end of a parameter or modifier list to indicate that any number of optional parameters or modifiers can be passed.

When you define a CA-Realizer procedure or function, you do not have to know the exact number of parameters and modifiers passed to it. In addition, procedures and functions can have an indeterminate number of parameters.

For example, alter the *Count3* procedure by making the *Increment* modifier optional:

```
PROC Count3(First, Last; ..)
  LOCAL k, Increment

  IF QNOptMods >= 1 THEN
    Increment = QOptMod(1)
  ELSE
    Increment = 1
  END IF

  FOR k = First TO Last STEP Increment
    PRINT k
  NEXT k
END PROC
```

Now the procedure has two required parameters and an unspecified number of optional modifiers. The *Increment* variable is set to the value of the first modifier; if no modifiers are supplied, the *Increment* variable is set to 1 by default.

You can now call *Count3* with either of the following statements:

```
Count3(StartAt, UpTo)
Count3(StartAt, UpTo; 2)
```

Also, if you use the *Count3* procedure with more than one modifier, the extra modifiers are ignored:

```
Count3(StartAt, UpTo; 1, 2, 5)
```

It may also be useful to have modifiers following optional parameters or to have optional modifiers as well. The following procedure has two required parameters, can take any number of additional parameters, and can take any number of modifiers:

```
PROC MyProc2(A, B, ..; ..)
END PROC
```

Note: Optional parameters and modifiers behave the same as required parameters and modifiers. However, since they do not have names, you must refer to them through the special functions `QNOptParams`, `QOptParam`, `QNOptMods`, and `QOptMod`.

Obtaining the Number of Optional Parameters and Modifiers

`QNOptParams` and `QNOptMods` are functions that return the number of optional parameters and modifiers passed to a procedure.

These numbers exclude the required parameters or modifiers. For instance, `QNOptMods` returns 0 when you invoke the *Count3* procedure with:

```
Count3(StartAt, UpTo)
```

`QNOptMods` returns 1 when you invoke *Count3* with:

```
Count3(StartAt, UpTo; 2)
```

Create another procedure called *MyProc* to see how different mixtures of optional parameters and modifiers affect the values returned by `QNOptParams` and `QNOptMods`:

```
PROC MyProc(A, B, ..; C, ..)
    PRINT QNOptParams, QNOptMods
END PROC
```


Here are some example calls to *MyProc* and the corresponding values returned by *QNOptParams* and *QNOptMods*:

Program Statement	QNOptParams	QNOptMods
MyProc(1, 3; 12)	0	0
MyProc(2, 7, 3; 9, 8, 8)	1	2
MyProc(1, 7, 3, 4; 2)	2	0

Execute *MyProc* with combinations of optional parameters and modifiers that you create.

Referencing Optional Parameters and Modifiers

Having determined the number of optional parameters or modifiers, access one of them using the *QOptParam* and *QOptMod* functions. These functions expect a number that indicates which optional parameter or modifier you want to reference.

For instance, *Count3* uses the following program statement to reference the first modifier:

```
Increment = QOptMod(1)
```

Now, add to the *MyProc* procedure by using *QOptParam* and *QOptMod* in a loop to access all the optional parameters and modifiers passed to *MyProc*:

```
PROC MyProc(A, B, ..; C, ..)
  LOCAL k

  PRINT QNOptParams, QNOptMods
  FOR k = 1 TO QNOptParams
    PRINT QOptParam(k)
  NEXT k
  FOR k = 1 TO QNOptMods
    PRINT QOptMod(k)
  NEXT k
END PROC
```

When you execute this procedure, the values contained in all the optional parameters and modifiers are printed. Try calling *MyProc* with several different parameter and modifier lists to see the results:

```
MyProc(1, 3; 12)
MyProc(2, 7, 3; 9, 8, 8, 0)
MyProc(1, 7, 3, 4; 2)
```

You can use optional modifiers and parameters to access a value or to set a value on the left side of an assignment. Add an index to the parameter or modifier if it is an array or a member name if it is a family.

You cannot refer to an optional parameter or modifier that does not exist. Therefore, a procedure or function that accepts optional parameters or modifiers should find out how many were passed to it with `QNOptParams` and `QNOptMods` before it refers to them. The `QVar` function is also helpful for determining what kind of variable was passed.

As an example, consider the following function, which takes a string and an indefinite number of strings to append to it:

```
PROC Phrases(string, ..)
  FOR i = 1 to QNOptParams
    PRINT string + QOptParam(i)
  NEXT i
END PROC

Phrases("A time to ", "laugh", "cry", "be born", \
  "die")
A time to laugh
A time to cry
A time to be born
A time to die
```

As a shortcut, use `__pn` (two underscores) in place of `QOptParam(n)` as an alias for the n^{th} optional parameter (for example, `__p1`, `__p2`). Likewise, you can use `__mn` instead of `QOptMod(n)`. This abbreviation requires that n be a constant value. For instance, both `__px+1` or `__p(x+1)` are illegal.

Redefining Procedures and Functions

When you define a procedure or function, it replaces any previous definition *within the same scope*. For example, each time you changed and executed the *Count3* procedure in the previous sections, the new definition replaced the old.

Note, however, that to redefine a procedure as a function, or a function as a procedure, you must first clear the procedure or function name with the `CLEAR` command:

```
CLEAR MyProc
```

Functions and Expression Evaluation

Expressions are evaluated left to right according to the order of operations. If you use a function that modifies a variable in the same expression as the variable itself, you may receive unexpected results:

```
FUNC Double(x)
    x = x * 2
    RETURN x
END FUNC

A = 5
PRINT A + Double(A)
20
```

Each operand in an expression is evaluated before it is operated upon. For instance, the *Double* function is called before the value of *A* is used in the addition. *Double* changes the value of *A*, and the new value is used in the addition, giving a result that you may not have intended.

Important! Caution is advised when using procedures and functions that change their parameters or modify global variables.

Recursion

In CA-Realizer, a procedure or function can call itself. This is known as *recursion*. Recursion is a very powerful language feature, and certain calculations lend themselves well to recursion. Recursion often allows you to produce a simple and easy solution, but this solution is not always the fastest or most efficient.

For example, the following function computes factorials using recursion:

```
FUNC Factorial(n)
  IF n <= 1 THEN
    RETURN 1
  ELSE
    RETURN n * Factorial(n-1)
  END IF
END FUNC
```

The initial value passed to *Factorial* determines the number of times it executes. *Factorial(5)* calculates $5 * 4 * 3 * 2 * 1$. This calls *Factorial* five times. Each time it executes, the value passed to the function is one less. When the value of its formal parameter reaches 1, the *Factorial* function returns a 1 without calling itself again. This special case prevents the *Factorial* function from recursing forever.

You must design a recursive procedure or function carefully. Always provide a way to terminate the recursion and use local variables to prevent naming conflicts.

Important! Use recursion with care, as it can take up a great deal of memory and time.

Creating and Running CA-Realizer Modules

You can break a CA-Realizer program into several files, each of which is called a *module* (or a *macro*). Modules often contain a set of procedure and function declarations. Proper use of module files can increase the clarity and structure of a program.

Creating Libraries

You can create entire libraries of procedures and functions that in turn can be used by any program that executes the RUN command with the appropriate files. This saves time by incorporating these existing program fragments in any new programs you build.

The best way to set up a library is to group procedures and functions in module files categorically—placing similar functions that relate to a single subject, design, or concept together. For example, you might create a module file for stock and bond functions, or special graphics procedures.

Using Modules as Procedures

You can also use modules like a procedure by passing parameters and modifiers and creating local variables. Procedures and functions declared within a module are not local to the module, but are local to the procedure or function that invoked the module. If the RUN command was not within a procedure, functions declared in a module would be global to the program.

Parameters and modifiers are passed to the module file by enclosing them within parentheses after the RUN command. Since there are no formal parameter names declared in a module, the parameters and modifiers are optional and are referenced the same way as optional parameters and modifiers used in procedures and functions.

Consider the following two files, MAIN.RLZ and SUB.RLZ. MAIN.RLZ contains the following statements,

```
A = "Test"
RUN "SUB"(1, A, 2)
```

while SUB.RLZ looks as follows:

```
PRINT QNOptParams, QNOptMods
PRINT QOptParam(1), QOptParam(2)
```

The output from running MAIN.RLZ is:

```
3  0
1  Test
```

Setting the Hourglass Cursor

If you have an application consisting of many modules that are being loaded, it is a good practice to use the SetHourglass command in your program to provide feedback to the user while modules are loading. You should use the ResetHourglass command when the modules have finished loading.

Using the EXECUTE Command

CA-Realizer's EXECUTE command allows you to build commands on the fly as a program runs. You can also use EXECUTE to change variables on the fly. EXECUTE runs any statement stored in a string.

For example, the following program permits a user to type in a formula for CA-Realizer to evaluate and then runs the formula directly through the CA-Realizer engine. Specifically, the user is prompted for the name of a member to add to a family:

```
INPUT "Enter a member name"; member
EXECUTE "MyFamily." + member + " = 1"
```

Because EXECUTE is an extremely powerful command, programmers often use it to create variables of family members at runtime, based on user input or database fields. Since it is possible to create a string which is identical to a CA-Realizer keyword, it is often worth checking to ensure that there are no likely syntax clashes by issuing the StrKeyword function as follows:

```
ReturnValue = StrKeyword(String)
```

This function determines whether a given string is identical to a CA-Realizer keyword, constant, command, or function name. It returns 1 if *ReturnValue* is a CA-Realizer keyword; otherwise, it returns 0.

Chapter 7

File I/O

In This Chapter	7-1
File Input and Output in CA-Realizer	7-1
Available File Commands and Functions	7-3
Basic File Concepts	7-4
Naming Files	7-5
Setting the Search Path	7-5
Manipulating Directories	7-7
Querying a File	7-9
Using the File Pointer	7-10
Using the Low-Level File Manipulation Commands and Functions	7-11
Opening a File	7-11
Writing to a File	7-12
Writing Text	7-13
Writing Numeric and Date-Time Values	7-15
Closing a File	7-15
Reading a File	7-15
Reading Text Data	7-17
Reading Numeric and Date-Time Values	7-18
A Complete File Read Example	7-19
Manipulating the File Pointer	7-20
Determining the Position of the File Pointer	7-20
Repositioning the File Pointer	7-21
Checking for the End of the File	7-22
Using the High-Level File Manipulation Commands and Functions	7-23
Importing/Exporting Data	7-24
Storing and Retrieving Sets of Variables	7-24

File Formats	7-26
Named	7-26
Plain	7-26
Importing and Exporting CA-Realizer Named Data	7-28
Importing and Exporting CA-Realizer Plain Data	7-29
Importing and Exporting Spreadsheet Named Data	7-30
Importing and Exporting Spreadsheet Plain Data	7-31
Importing and Exporting Text Named Data	7-31
Importing and Exporting Plain Text Data	7-33
Binary Files	7-34
Importing and Exporting XBase Data Files	7-37
Querying a .DBF File	7-39
Importing .DBF Files	7-40
Exporting .DBF Files	7-41

Chapter 7

File I/O

In This Chapter

This chapter describes how to read, write, and save data in files using CA-Realizer. Also discussed are basic file manipulation concepts, file formats, and the various CA-Realizer file I/O commands and functions.

File Input and Output in CA-Realizer

In BASIC programming, three different file organization and accessing schemes are usually referenced—*sequential*, *random access*, and *binary*.

Sequential

Sequential files store data as lines of text; each line—a *record*—is comprised of data items—*fields*—separated by some common delimiting character (for example, a comma or a tab). Typically, each record is terminated by a carriage return and line feed character (or CRLF, ASCII characters 13 and 10, respectively).

Random Access

Random access files save data as fixed-length records. Like sequential files, each line in a random access file is considered a *record* and its data items *fields*. However, because its records are *fixed-length*, delimiters are not necessary to separate fields and records.

Binary

In binary files, any file is treated as a numbered sequence of bytes, without regard to record sizes, ASCII characters, or delimiters. Binary access is therefore different from random access, where the number of bytes is fixed by the predefined length of a record.

Historically, this distinction into three different file organization and accessing schemes was necessary because earlier versions of BASIC used different commands for manipulating these three file organization schemes (for instance, WRITE #, PRINT #, and INPUT # for sequential file manipulation; GET, PUT, and FIELD for random access files; and GET\$, PUT\$, and SEEK for binary files).

CA-Realizer blurs the traditional distinction between the three BASIC file organization and access types. (In fact, the above commands have no direct syntactic equivalents in CA-Realizer BASIC.) Instead, CA-Realizer distinguishes between two types of files and the commands and functions you can use to access them—*low-level* commands and functions and *high-level* commands and functions.

Low-Level Commands and Functions

Low-level commands and functions are similar to those in other languages, such as C, and provide you with the most power and flexibility. However, they also require you to do the most work. Therefore, for most applications, you will find that high-level access is the more convenient.

High-Level Commands and Functions

With CA-Realizer's high-level commands and functions, you can use the same set of functions for storing and reading data in a variety of file formats, including those of popular application software. It distinguishes between them by providing different modifiers.

As a result, you will find that when you compare CA-Realizer BASIC to earlier versions of BASIC (and indeed most other languages), CA-Realizer is both more powerful and more flexible in the ways it allows you to store and manipulate your data.

Available File Commands and Functions

The following table summarizes all of the CA-Realizer file commands and functions:

Command	Description
FileClose	Closes an open file.
FileDB	Imports and exports data from XBase files (.DBF).
FileDelete	Deletes a specified file. (KILL is a synonym for FileDelete.)
FileEOF	Determines if the end of the file has been reached.
FileExport	Writes data in a variety of file formats.
FileImport	Reads data in a variety of file formats.
FileMakeName	Returns a unique file name.
FileOpen	Opens a file for reading and/or writing.
FilePos	Returns the current position of the file pointer.
FileQ	Returns information about a file, without first having to open it.
FileQUnique	Returns a unique file number. (FREEFILE is a synonym for FileQUnique.)
FileRead	Reads data from an open file into a variable.
FileSeek	Sets the position of the file pointer within an open file.
FileWrite	Writes data to an open file.
LOF	Returns the size of a file.
RENAME	Renames a file.

Basic File Concepts

Like most software applications, the data files created by CA-Realizer are written to disk and are each distinguished by a unique name (called the *file name*).

To access the information in a CA-Realizer disk file, you must usually open the file first (note that this is not necessary when accessing files using the high-level commands, FileImport, FileExport, and FileDB). When a file is opened, it is assigned a unique number so that you can easily work with several files at once.

Once a file is opened, you can either read from it or write to it. When you are finished with a file, close it to ensure any changes you have made are saved to disk.

In general, you can create and manipulate data files with the following CA-Realizer commands:

Group	File Commands
Low-Level	FileOpen
	FileWrite
	FileRead
	FileClose
	FilePos
	FileSeek
High-Level	FileDB
	FileImport
	FileExport

Naming Files

Names and Extensions All CA-Realizer file names adhere to the DOS naming conventions. That is, file *names* can be up to eight characters in length and file *extensions* can be up to three characters in length. In addition, these names cannot contain spaces or other special characters.

The file extension typically indicates the type of information the file contains. For instance, the .DAT extension suggests a data file, and the .RLZ extension marks CA-Realizer program files.

REALIZER.EXE, SAMPLE.DAT, and POKER.RLZ are examples of valid file names.

Paths

Since you can find data files in any of the directories on your disk, file names can also include a *path*. This path indicates the drive and/or directory in which the file is stored.

For example, the following command opens the CORP.DAT file in the STOCKS directory on the C drive:

```
FileOpen(1, "C:\STOCKS\CORP.DAT")
```

Setting the Search Path

If you do not specify a complete path for a file name when attempting to open or save a file, CA-Realizer looks for the file in the directories indicated by the *search path*. The search path is a string variable preset by the CA-Realizer environment. (This type of variable is called a *system variable*.)

A path is a list of semicolon-separated drive and directory names using the standard DOS naming convention of *drive:directory name*. By default, the search path is set to the current directory. You can change the paths defined in the search path string using the SetSys or AddSys commands with one of the CA-Realizer constants: _LoadDir or _MacroDir.

_LoadDir

The `_LoadDir` constant tells CA-Realizer where to look when loading either data files or bitmap files. CA-Realizer searches the specified paths in the order in which they are listed. A semicolon must be used to separate each of the paths defined in this string.

Note: You can also use the standard DOS period (.) and double-period (..) characters in the search path to search the current directory and the parent of the current directory, respectively.

For example, the following command instructs CA-Realizer to look in `REALIZER`, `REALIZER\MANUAL`, `REALIZER\LIB`, and current directory—in that order:

```
SetSys(_LoadDir, "\REALIZER;\REALIZER\MANUAL;" + \\
"\REALIZER\LIB;.")
```

_MacroDir

Similarly, the `_MacroDir` constant instructs CA-Realizer where to look for macros loaded with the `RUN` command or external DLLs.

Adding Directories to an Existing Path

It is a good idea to add directories to the beginning or end of an existing path instead of simply replacing the path. For example:

```
NewPath = "C:\MYDIR"
SetSys(_LoadDir, QSys(_LoadDir) + ";" + NewPath)
```

The `AddSys` command can also be used:

```
AddSys(_LoadDir, "C:\MYDIR")
```

For more information on file names and paths, consult your DOS, Windows, and/or OS/2 documentation.

Manipulating Directories

CA-Realizer provides a number of useful commands and functions for manipulating directories. They are summarized in the following table and are described in greater detail below:

Command	Description
QDir	Returns the current working directory for the current drive.
CHDIR	Changes the current directory.
SetDir	Changes the current directory.
MKDIR	Creates a new directory.
RMDIR	Removes a directory.
FILES	Generates a list of files in a directory.

QDir

The QDir function returns the current working directory for the current drive:

```
Directory = QDir [(Drive)]
```

You can optionally specify a drive to return the current working directory for that drive instead.

CHDIR and SetDir

You can change the current directory with the CHDIR command or the SetDir command:

```
SetDir(Path [, _ChangeDrive])
```

```
CHDIR Path
```

If the directory supplied in SetDir contains a path specification, you can also change the current drive by specifying the optional _ChangeDrive parameter.

MKDIR and RMDIR

Like their DOS equivalents, MKDIR creates a new directory, while RMDIR removes a directory:

```
MKDIR DirectoryName
```

```
RMDIR DirectoryName
```

FILES

You can use the FILES command to generate a list of files in a directory. The general syntax of FILES is:

```
FILES Filter  
Array = FILES(Filter)
```

Use *Filter* to specify the path (that is, the drive and directory) in which you want the program to look, and the *file specification*. A file specification is used to identify the file (or files) for which you want to search and can include the wild card characters * and ?.

Several examples are presented below.

The following statement lists all file names in the current directory with the .DAT file extension:

```
FILES "*.DAT"
```

The following statement lists all file names in the C:\DATA directory that begin with TEST and have any file extension (for example, TEST01.DBF or TESTA.RLZ):

```
FILES "C:\DATA\TEST*.*"
```

The following statement lists all file names in the current directory regardless of name or file extension:

```
FILES "*.*"
```

Note: The default filter is *.* (all files).

FILES generates a list of file names that are currently stored in the specified directory and match the given file specification. When you use FILES as a command, it displays the matching files in the Print Log. When you use it as a function, it returns the list of files as an array of strings. (If there are no files that match the file pattern, FILES returns a scalar value of 0.)

Querying a File

CA-Realizer permits you to receive information about a closed or open file with the FileQ and LOF functions, respectively.

FileQ

The FileQ function allows you to determine whether a file exists, as well as its size and the date and time it was last modified. Unlike LOF, the file does not have to be open.

```
Value = FileQ(FileName, Action)
```

FileName is a string holding a file name. (If the *FileName* string does not include a path, FileQ looks for the file in the first directory of the search path.)

Action is a constant indicating the type of information to retrieve (such as *_Size* to return the file's size in bytes or *_Exists* to check if a file exists). For example, the following line prints the size in bytes of a file called TEST.DAT:

```
PRINT FileQ("TEST.DAT", _Size)
```

Enter and run the following example. The *FileInfo* procedure looks for a given file name and prints information about the file if it is found:

```
' Manual\PG_07\FILEINFO.RLZ

PROC FileInfo(FileName)
  IF FileQ(FileName, _Exists) THEN
    PRINT USING "File found: &;" \\
      FileQ(FileName, _Name)
    PRINT USING "Last modified on D(D5, M4 "+ \\
      "D3, Y2) at D(h4:m1 h5)"; \\
      FileQ(FileName, _DateTime), \\
      FileQ(FileName, _DateTime)
    PRINT USING "Size: P(0) Bytes"; \\
      FileQ(FileName, _Size)
  ELSE
    PRINT "File not found."
  END IF
END PROC

INPUT "Enter filename:", "File Info"; FileName
FileInfo(FileName)
```

LOF

If a file is already open, its size can be determined with the LOF function as follows:

Size = LOF(*FileNum*)

FileNum represents the number of the open file to query.

Using the File Pointer

As you read from—or write to—a file, the *file pointer* keeps track of where you are in the file. The file pointer always points to the next character to be read.

As data is read or written, the file pointer advances automatically. However, you may want to examine only a certain portion of data in a file, or search for a particular record, because a data file does not always have relevant information in every single block. In these cases, you can move the file pointer to read and write at the desired location.

CA-Realizer provides a set of functions that allow you to manipulate the file pointer—for example, you can determine the current position of the file pointer or if the end-of-file (EOF) has been reached, skip through the file without reading or writing over the data, or change its current position. See *Manipulating the File Pointer* later in this chapter.

Using the Low-Level File Manipulation Commands and Functions

The low-level commands and functions provided by CA-Realizer allow you to perform basic file manipulation tasks, such as opening files, reading and writing data to and from files, and closing files, as well as manipulating the file pointer.

Opening a File

FileOpen

Before you can manipulate data in a file using the low-level file commands, you must open the file using the FileOpen command as follows:

```
FileOpen(FileNum, FileName, Permission)
```

FileNum is the number to assign to the file. You must assign each file a unique number. Note that you can use the FileQUnique function to have CA-Realizer choose an unused file number for you.

FileName is a string containing the file name, and *Permission* is a constant that allows you to restrict how the file can be accessed—for example, whether it should be opened as “read-only” or “read-write.”

Note: By default, FileOpen uses `_Read` access to prevent data from being changed accidentally. In addition, when you use the `_Write` and `_Append` constants, they create a file if one does not already exist.

For example, these statements first open the SAMPLE.DAT file (identified as file number 10) and then create a second file called TEST.DAT (both files can be written to):

```
FileOpen(10, "SAMPLE.DAT", _Write)
MyFile = FileQUnique
FileOpen(MyFile, "TEST.DAT", _Write)
```

When these two FileOpen commands execute, the SAMPLE.DAT and TEST.DAT files are created.

Writing to a File

FileWrite

Using the FileWrite function, you can write data to an open file. By default, FileWrite writes numeric and date-time values in a binary format according to their type, and strings as ASCII characters.

Note: FileRead and FileWrite manipulate values in the CA-Realizer format, which means that each character in a string occupies a single byte, and both numerics and date-time values take 8 bytes each for a scalar or array element.

Every time you execute FileWrite, CA-Realizer places data values at the location indicated by the file pointer; the file pointer is then advanced to indicate the location after the last carriage return added.

Note: Data written with FileWrite is easily read with FileRead.

The general form of FileWrite is as follows:

```
FileWrite(FileNum, Data [, Count [, AlphaSize]])
```

FileNum is the file number (as specified in FileOpen) of the file to which data should be written. *Data* represents the information to be written to the file indicated by *FileNum*. *Data* can be a variable, constant, or expression of any type. (If the value is an array, all the elements of the array are transferred to the file.)

Count is the number of characters (for string values) or array elements to write. Use *AlphaSize* when writing an array of strings, in which case *AlphaSize* characters of each string element are written to disk without CRLFs.

Note: Each FileWrite transfers a *single* scalar or array. Therefore, use several FileWrite commands to save more than one variable or value.

To demonstrate how the `FileWrite` command works, create the following data set. These variables contain test results from a college Physics class:

```
ClassName = "Physics 104"
TestDate = StrToDate("3/25/93")
StudentNumber = 10
Scores = {95, 98, 56, 72, 88, 93, 100, 67, 91, 84}
```

Now use the following commands to transfer the contents of these variables to the `TEST.DAT` file:

```
FileWrite(MyFile, ClassName)
FileWrite(MyFile, TestDate)
FileWrite(MyFile, StudentNumber)
FileWrite(MyFile, Scores)
FileClose(MyFile)
```

The three scalars and the elements of the *Scores* array are written to the file. Each value and each array element is separated by a carriage return.

Writing Text

By default, when *Data* is a string value, the `FileWrite` function saves the data as ASCII characters.

Note: When you automatically terminate strings with carriage returns, the termination is actually a 2-byte CRLF combination.

A Single Line

If *Data* holds a single line of text, the value specified by the optional *Count* parameter affects how much text is written to file. If *Count* is omitted, the entire string is written, followed by a CRLF character. If *Count* is specified, exactly that number of characters are written and no CRLF is added. (If *Count* is greater than the size of the *Data* string, spaces are appended to the end of the string.)

For example, suppose you opened the file named `MOREDATA.DAT` for writing as file number 5. If *A* = "She sells sea shells by the sea shore" and *B* = "She does", then the following command writes "She sells sea shells by the sea shore" followed by a CRLF character:

```
FileWrite(5, A)
```

On the other hand, “She sells She does” is written by the following:

```
FileWrite(5, A, 10)
FileWrite(5, B)
```

This is because the first FileWrite command wrote “She sells ”, but the next one appended “She does”.

Note: When only a portion of a string is transferred to a file, no carriage return is included. This is useful for producing files with fixed-size records.

An Array of Strings

FileWrite operates slightly differently when *Data* is a string array. If you do not specify *Count*, each element of the array is written completely, followed by a CRLF character. If you do specify *Count*, only that number of *elements* are written in full, followed by a CRLF character.

Try the following three FileWrite combinations—the results are compared later when you use FileRead to read this data back in:

```
C[1:5]={"She ", "sells ", "sea ", "shells ", "by "}
C[6:8]={"the ", "sea ", "shore"}
MyFile = FileQUnique
FileOpen(MyFile, "MYSTUFF.DAT", _Write)
FileWrite(MyFile, C)
FileWrite(MyFile, C, 4)
FileWrite(MyFile, C, 4, 2)
FileClose(MyFile)
```

Remember, to write only a portion of either the characters in a string or the elements in an array, just add the *Count* parameter to the FileWrite command. For example, to write the first five elements of the *Scores* array, use the following command:

```
FileWrite(10, Scores, 5)
```

You can also write the first eight characters of a string variable:

```
FileWrite(10, ClassName, 8)
```


Writing Numeric and Date-Time Values

By default, CA-Realizer writes numeric and date-time values in a *binary* format according to their type. Therefore, to write numbers or dates as ASCII characters, first convert them to strings using `Sprint` or `NumToStr`.

Closing a File

FileClose

When you finish working with an open file, close it with the `FileClose` command:

```
FileClose(FileNum)
```

For example:

```
FileClose(10)
FileClose(MyFile)
```

`FileClose` assures that all the information sent to the file is actually transferred to disk. When CA-Realizer executes the `FileClose` command, any remaining data is flushed out of temporary storage and into the disk file.

Note: When you close a file, its file number is no longer defined and you can reuse it again.

Reading a File

FileRead

Once a file is opened, you can use the `FileRead` command to read data from it. The syntax for `FileRead` is almost identical to that for `FileWrite`, except that you can optionally include an additional parameter, *Type*:

```
FileRead(FileNum, Data [, Count
        [, Type [, AlphaSize]])
```

As in `FileWrite`, *FileNum* is the file number, *Data* is a variable, and *Count* is the number of characters to read.

Type is a CA-Realizer constant that describes the type of data to read; it can take the values `_Alpha`, `_Real`, or `_DateTime`.

Use *AlphaSize* when reading an array of strings. It specifies the number of characters to be read from disk from each string *element*, without CRLFs.

Note: You can also use the `FileImport` and `FileExport` commands—discussed later in this chapter—for higher level file access.

Reading a Single Line The following example demonstrates how to read single lines from a file. It first opens the `TEST.DAT` file—created earlier—with read permission. (Note that since the *MyFile* variable is no longer defined, you can reuse it to open the file.) Each of the `FileRead` statements reads data until it reaches a carriage return:

```
MyFile = FileQUnique
FileOpen(MyFile, "TEST.DAT", _Read)
FileRead(MyFile, ClassName, 1, _Alpha)
FileRead(MyFile, TestDate, 1, _DateTime)
FileRead(MyFile, StudentNumber, 1, _Real)
```

FileNum and *Data* match the first two parameters of their `FileWrite` counterparts—*FileNum* represents the file number and *Data* represents the data to be read from *FileNum*. CA-Realizer places the data values read from the file in these variables.

Count indicates the number of data values to read from the file. Since this parameter is 1 in each of these three program lines, CA-Realizer reads the first three values as scalars, and the file pointer advances after each command. If this parameter is greater than 1, CA-Realizer reads the specified number of data elements into an array, and the file pointer advances to the data value after the given number of elements.

Type describes the type of data to read, as specified by the type constants `_Alpha`, `_Real`, or `_DateTime`.

Note: CA-Realizer reads values in the same order as they were written.

Reading Multiple Lines Read the remaining test score data from the TEST.DAT file to rebuild the *Scores* array. The number of data elements in this array is indicated by the *StudentNumber* variable. This value is used in the *FileRead* command to read the entire *Scores* array:

```
FileRead(MyFile, Scores, StudentNumber, _Real)
```

When you are finished reading the file, close it as follows:

```
FileClose(MyFile)
```

Reading Text Data

Portions of a String If you do not include *Count* and *Type* to indicate the number and type of data you want to read, CA-Realizer assumes you want to read a single line of text. However, if you include *Count* without *Type*, CA-Realizer reads *Count* number of characters. In this case, carriage returns are read like any other character, and the file pointer advances automatically.

For example, the following reads ten characters from the *Label* variable in *OtherFile*, and then reads the next eight characters:

```
FileRead(OtherFile, Label, 10)
FileRead(OtherFile, Label, 8)
```

The following example first assigns a unique file number to MYSTUFF.DAT using the *FileQUnique* function. It then opens the file and issues the *FileRead* command three times to read the first 53 characters, the first 29 characters, and the first 16 characters of MYSTUFF.DAT, respectively:

```
MyFile = FileQUnique
FileOpen(MyFile, "C:\REALIZER\MANUAL\MYSTUFF.DAT", \
_Read)
FileRead(MyFile, A, 53)
FileRead(MyFile, B, 29)
FileRead(MyFile, C, 16)
FileClose(MyFile)
PRINT A
PRINT "-----"
PRINT B
PRINT "-----"
PRINT C
```

The results appear as follows:

```
She  
sells  
sea  
shells  
by  
the  
sea  
shore  
-----  
She  
sells  
sea  
shells  
-----  
Shsesesh
```

An Array of Strings

If *Type* is set to `_Alpha` in the `FileRead` command, *AlphaSize* is used to control how strings are read:

- If you do not specify *AlphaSize*, each element is read up to a carriage return and *Count* strings are read until the end of the file is reached. An error results if *AlphaSize* is specified, and there are not enough characters left in the file..
- If you specify *AlphaSize*, CA-Realizer uses it as the number of characters to read into each element. (Carriage returns and line feeds are read like any other character.)
- If *AlphaSize* is an array, it should contain *Count* elements indicating the number of characters to read into each corresponding string.

Reading Numeric and Date-Time Values

You can use `FileRead` to read numeric and date-time values. Simply set *Type* to either `_Real` or `_DateTime`, depending on the type of data to be read. The type of *Data* is set accordingly.

For example, the following statement reads one real value:

```
FileRead(1, age, 1, _Real)
```

CA-Realizer then reads and writes files using its internal numeric and date-time format, which is much faster than reading and converting values to text.

When you specify *Type*, *Count* indicates the number of values to read. (If *Count* is 1, *Data* is a scalar; if *Count* is greater than 1, *Data* is an array with *Count* elements.) This is a very fast way of reading a list of reals or dates into an array. If there are less than *Count* elements left in the file, an error occurs.

A Complete File Read Example

Here are several `FileRead` examples used together. The command sequencing used here is often found in the body of file input procedures:

```

FileName = StdOpen("*.DAT")
IF FileName <> "" THEN
    MyFile = FileQUnique
    FileOpen(MyFile, FileName, _Read)
    FileRead(MyFile, ClassName, 1, _Alpha)
    FileRead(MyFile, TestDate, 1, _DateTime)
    FileRead(MyFile, StudentNumber, 1, _Real)
    FileRead(MyFile, Scores, StudentNumber, _Real)
    FileClose(MyFile)
END IF

```

Run the program and select the TEST.DAT file name from the Open File dialog box—created with the `StdOpen` function. (For more information about the `StdOpen` function, see the *CA-Realizer Language Reference Guide* and “Controlling Input and Output” in this guide.) To confirm that all the data has been read from the file properly, execute the following `PRINT` commands to display the variables in the Print Log:

```

PRINT "Data from file: "; FileName
PRINT ClassName; "    Test of "; TestDate
PRINT "Number of Students: "; StudentNumber
PRINT "Scores: "; Scores

```

You can also look in the Variable Display to see defined variables and their contents. Press `CTRL+V` or select `SHOW VARIABLES` from the `RUN` menu.

Manipulating the File Pointer

When using the `FileWrite` and `FileRead` commands, the file pointer is advanced automatically. By executing a series of these commands, you can read or write an entire file.

However, you may want to examine only a certain portion of data in a file, or to search for a particular record. To skip from place to place in a file without reading or writing over the data, use the `FilePos` function to determine the current file position, and the `FileSeek` command to change the current position.

Determining the Position of the File Pointer

`FilePos`

The `FilePos` function determines the current position of the file pointer for any open file. This position is measured in bytes from the beginning of the file, starting at 0:

```
PointerLocation = FilePos(FileNum)
```

As an example, open the `TEST.DAT` file again. The following lines determine the position of the file pointer after the first variable is read:

```
FileOpen(MyFile, "TEST.DAT", _Read)
FileRead(MyFile, ClassName, 1, _Alpha)
FilePointer = FilePos(MyFile)
FileClose(MyFile)
PRINT FilePointer
13
```

The `ClassName` string takes up 13 bytes of the file. The string has 11 characters and terminates with a carriage return and line feed combination.

Since each file has its own file pointer, you must supply a file number as a parameter to the `FilePos` function.

Repositioning the File Pointer

FileSeek

The FileSeek command allows you to set the position of the file pointer. Its general form is as follows:

```
FileSeek(FileNum, Amount [, Where])
```

FileNum is a file ID of a file opened with FileOpen, and *Amount* is an integer value—measured in bytes—representing the number of bytes the file pointer should be moved (it can be positive or negative).

By default, the file pointer is moved relative to its current location. You can optionally include *Where* (which takes the values `_Beg`, `_Cur`, or `_End`) to specify the position the file pointer should be moved from (the beginning of the file, the current position, or the end of the file, respectively).

Note: You can also use the SEEK command.

Suppose a data file includes fixed-size records, each 32 bytes long, with four fields, each 8 bytes in size. You can move the file pointer directly to the start of any record using the FileSeek command. For example, to read data from the fifth record, use the following command:

```
FileSeek(MyFile, 4 * 32, _Beg)
```

It moves the file pointer 128 bytes from the beginning of the file.

Now read the second field in this record. First move the file pointer 8 bytes relative to the current position, and then use the FileRead command:

```
FileSeek(MyFile, 8, _Cur)
FileRead(MyFile, Field2, 8)
```

By moving the file pointer to the end of the file and obtaining its position with FilePos, you can determine the length of a file:

```
FileSeek(MyFile, 0, _End)
FileLength = FilePos(MyFile)
```

You can also move the file pointer beyond the end of a file to leave blank space that you can fill in later.

Checking for the End of the File

Sometimes the complete structure and size of a data file is unknown. While the file pointer can be moved beyond the end of the file and it is possible to write beyond the end of the file, an error is received if you attempt to read past the end of the file.

FileEOF

To prevent reading past the end of a file, use the FileEOF function before a FileRead command. It checks where the file pointer is currently located, and returns a 1 if it is at the end of the file (otherwise, FileEOF returns 0).

The general form of FileEOF is as follows:

```
Value = FileEOF(FileNum)
```

Suppose the only thing you know about the TEST.DAT file is that it starts with a string and a date-time value, followed by an undetermined number of real numbers. The following would read all the data from this file:

```
FileOpen(MyFile, "TEST.DAT", _Read)
FileRead(MyFile, AlphaVar, 1, _Alpha)
FileRead(MyFile, DateVar, 1, _DateTime)
WHILE (NOT FileEOF(MyFile))
    FileRead(MyFile, Data, 1, _Real)
    PRINT Data
END WHILE
FileClose(MyFile)
```

As long as the FileEOF function does not find the end of the file, the FileRead loop continues reading values and advancing the file pointer.

Using the High-Level File Manipulation Commands and Functions

The file commands and functions discussed so far deal with the contents of files in bits and pieces. If a file has a record structure, your program needs to decode the data format to read through all the records, leaving you the burden of adding all of the low-level file manipulation code into your program.

However, the CA-Realizer high-level commands—FileDB, FileImport, and FileExport—relieve you of this burden. These commands quickly and conveniently save and retrieve data created by other applications and in a variety of formats, including CA-Realizer, other text editors, CA-Compete!, CA-SuperCalc, Microsoft Excel, Lotus 1-2-3, and XBase-compatible database managers.

Each command, presented in greater detail later in this chapter, is similar syntactically:

```
family = FileDB(File, Type, Action
               [, Start [, Count]][, Info, Data] )
FileImport(File, Type, Format [, Template]
           [, Var1 [..., VarN]][, Start [, Count ]])
```

In general, *File* is the name of the file to which data should be written or read from, *Type* indicates the type of file to read (for example, whether it is a CA-Realizer or CA-Compete! file), and *Format* specifies the format of the data within the file (*_Named* or *_Plain*).

You can use *Start* and *Count* modifiers to modify the portion of the data read, as described earlier. Also, note that *Type*, *Action*, and *Format* take on specific, CA-Realizer constant values.

Var1 through *VarN* are the variables to write to or read from a file. They can be any combination of scalars, vectors, or families whose members are scalars or vectors. Records, subfamilies, arrays of families, and arrays of arrays are not allowed. Portions of these complex data types, can, however, be assigned to other variables and then exported.

Importing/Exporting Data

To access information in a disk file, or to write data to a disk file, use the `FileImport` and `FileExport` commands. They allow you to read and write data in a variety of file formats.

`FileExport`

The simplest use of the `FileExport` command is saving CA-Realizer variables—and their associated information—to a file. You can save data to many different types of files in a variety of file formats, including the internal CA-Realizer format, standard spreadsheet formats, and custom binary formats.

`FileImport`

In addition, data from these various files and data formats can be imported into CA-Realizer using the `FileImport` command, including information written with `FileExport`.

`FileDB`

Use this command to read and write data from XBase-compatible files.

Storing and Retrieving Sets of Variables

Unlike the low-level `FileRead` function, both `FileDB` and `FileImport` are designed to read *sets* of variables at once. The fields form columns of data that are read into CA-Realizer array variables.

Consider the following variables:

```
Phrase = "Do not disturb"
Days[2:4] = {StrToDate("03/07/82"), \
             StrToDate("08/24/85"), StrToDate("12/01/89")}
Amount = 1450.00
Names[1:6] = {"Tom", "Dick", "Harry", "Mary", \
              "Jane", "Ann"}
Values[3:6] = {12, 34, 56, 78}
```

These variables, written in a spreadsheet or database format, would appear as follows:

Row	Phrase	Days	Amount	Names	Values
1	Do not disturb		1450.00	Tom	
2		03/07/82		Dick	
3		08/24/85		Harry	12
4		12/01/89		Mary	34
5				Jane	56
6				Ann	78

Each row represents a *record* of data; each item in a record, whose names are indicated by the column headings, is also referred to as a *cell*. Scalar values fill only the first record; arrays fill a number of records corresponding to their valid range.

The FileDB, FileImport, and FileExport formats all work similarly with this concept of records. Using these commands, you can write CA-Realizer data in this tabular format (where the variables fill the cells in this fashion), as well as read data from a spreadsheet or database (each column becomes a variable and the valid range is determined by the rows in the column that have entries).

FileExport always writes all the elements in a variable to a file. FileImport normally reads all the data records, but it can also operate on portions of the data. If you specify *Start*, it determines the record number at which to start reading. When you supply *Count*, it indicates the maximum number of records to read from the file. If the portion of a column requested by FileImport has no valid entries, an error results.

FileDB works in a similar manner to the other two commands and uses similar parameters. However, the FileDB function is capable of *both* importing and exporting XBase-compatible .DBF files.

To help avoid confusion, note that:

- FileDB does not take the *Format* parameter. Since its format is fixed, it reads and writes .DBF files.
- FileImport and FileExport do not take *Action* parameters, since their action is fixed (either importing or exporting files).

File Formats

This section provides information about the various file formats that can be used with CA-Realizer's high-level file-manipulation commands when reading and writing data.

Named

CA-Realizer provides a *Named* format, which is the only format that preserves *all* the format information and keeps scalars as scalars. You can import and export Named data.

Plain

The *Plain* format is similar to the Named format, but there are no variable names at the top of the data columns. Instead, a template, defined by the family *Template*, indicates how the file is organized.

```
FileExport(File, Type, _Plain, Template)
FileImport(File, Type, _Plain, Template
           [, Start [, Count]])
```

Template specifies the variables to read and their formats.

This family should have two members: *VARIABLE* and *FORMAT*. There should be an entry in each of these members for every column in the file. If *FORMAT* is a scalar, the format it specifies is used for all variables.

The *VARIABLE* member is a string array containing the name of the variables to fill for each column in the file. It can be a family member, array, or scalar. Blank entries in *VARIABLE* indicate the column should be skipped.

FORMAT contains an alpha array with entries of the form *f*[*mm*] describing the layout of each column. *f* can be one of the following column layout codes:

Code	Meaning
A	Alpha (string)
D	Date-time
R	Real (numeric)
X	Skip
Z	Format as found

mm is the optional width of the field in characters, ranging from 0 to 1024. For example, the variables *Name*, *Age*, and *Birthdate* can be exported with the following template:

```
Template.VARIABLE = {"Name", "Age", "Birthdate"}
Template.FORMAT = {"A20", "R", "D"}
```

For all the plain formats (except those for the spreadsheet files), *File* can be the number of a file opened with *FileOpen* instead of a file name. This allows *FileImport*(*_Plain*) and *FileExport*(*_Plain*) to be mixed with *FileRead*, *FileWrite*, and *FileSeek* commands.

Importing and Exporting CA-Realizer Named Data

The formats for the `FileImport` and `FileExport` commands for named CA-Realizer data are:

```
FileExport(File, _Realizer, _Named  
           [, Var1 [..., VarN]]  
  
FileImport(File, _Realizer, _Named  
           [, Var1 [..., VarN]][; Start [, Count]])
```

`FileExport(_Realizer,
_Named)`

`File Export(_Realizer, _Named)` takes a list of variables specified by *Var1 ... VarN* and writes them to *File*, using CA-Realizer's internal format. The list of variables can include any combination of scalars, arrays, records, entire families, and family members. This command stores the valid range of arrays as well.

`FileImport(_Realizer,
_Named)`

`FileImport(_Realizer, _Named)` reads and restores the variables specified as *Var1 ... VarN*. If you do not specify any variable names, all the variables in the file are imported. If you try to import a variable that is not in the file, an error results.

Start and *Count* can be used to read a portion of array values from the file. *Start* indicates the starting row to import, and *Count* indicates the maximum number of rows to read.

Note: The `_Realizer _Named` format is the only one that preserves *all* the format information and keeps scalars as scalars.

Importing and Exporting CA-Realizer Plain Data

`FileExport(_Realizer,
_Plain)`

You can read and write data in the CA-Realizer *Plain* format, without the associated variable names and related information, by using the `FileExport(_Realizer, _Plain)` format.

```
FileExport(File, _Realizer, _Plain, Template)
FileImport(File, _Realizer, _Plain, Template
           [, Start[, Count]])
```

Template specifies the variables to read and their formats. *Start* and *Count* can be used to read a section of a file. *Start* indicates the starting row to import, and *Count* indicates the maximum number of rows to read.

`FileImport(_Realizer,
_Plain)`

The CA-Realizer Plain format allows greater control over the data than the CA-Realizer Named format. Since variable names are not embedded in the file, data can be read and assigned to a different variable name than the one written. Each record in the file is the same size, so a program can skip over entire records with `FileSeek` and import data from the middle of a file.

The only formats allowed with Plain are *An* (a string *n* characters long), *R* (a numeric value), and *D* (a date-time value). (The *X* and *Z* formats are not allowed.) String fields require a specified width in characters large enough to hold the largest single element in that column.

To skip an entire column of data when the data is imported, use `""` for the *VARIABLE* field of that column.

The values are stored using the CA-Realizer internal format. The size of a record is the sum of the sizes of each of its fields. Date-time and numeric values are 9 bytes each. String fields are 3 bytes more than the number of characters in the field. The size of a record can also be determined by opening a file, importing one record using the proper template, and looking at the file position returned by the `FilePos` function.

Importing and Exporting Spreadsheet Named Data

Exporting

You can use the `FileExport` Named formats to write CA-Realizer data to CA-SuperCalc, CA-Compete!, Microsoft Excel, or Lotus 1-2-3 spreadsheets. The general form is:

```
FileExport(File, Type, _Named [, fTemplate]  
          [, Var1 [... , VarN]])
```

When the file is read later into the spreadsheet application:

- Each exported CA-Realizer variable becomes a column in the spreadsheet.
- The first row lists the names of the variables.
- The rest of the rows contain the data.
- Scalar values fill only the second row.
- Arrays occupy the number of rows corresponding to the valid range of the array. For instance, an array valid for elements 3 through 8 has values in the spreadsheet in rows 4 through 9 of its column.

Importing

You can use `FileImport(Type, _Named)` to read spreadsheet files. The general form is:

```
FileImport(File, Type, _Named. [, fTemplate]  
          [, Var1 [... , VarN]] [, Start [, Count]])
```

File is the file name and *Type* is the type of file to read; *Type* can be `_SuperCalc`, `_Compete`, `_Excel`, or `_Lotus`; *Template* specifies the variables to read (and their formats); and *Start* and *Count* can be used to read a section of a file (*Start* is the starting row to import, and *Count* is the maximum number of rows to read).

It is assumed that each column represents a variable and the first valid row of the spreadsheet contains the variable names. Columns without valid variable names and with duplicate names are skipped. Note that a variable's type is determined by the data. All variables are read in as arrays, even if there is only one element. In the column, any element that does not match the data type of the first element is set to a default value (0 for reals, 1/1/1900 for date-time values, and "" for strings). The `_Lotus` format does not support date-time values.

Importing and Exporting Spreadsheet Plain Data

Use `FileImport _Plain` formats to read in spreadsheet data without a heading, or data that would be improperly typed using the rules presented in the `Spreadsheet _Named` format.

```
FileExport(File, Type, _Plain, Template)
FileImport(File, Type, _Plain, Template
           [, Start [, Count]])
```

File is the file name; *Type* is the type of file to read and can be `_Compete`, `_SuperCalc`, `_Excel`, or `_Lotus`; and *Template* specifies the variables to read and their formats.

In `_Plain` spreadsheet formats, the *Count* portion of the field format is ignored. If you use the `Z` format, the type of the imported data is determined automatically. Therefore, unless the normal type of a variable needs to be overridden (such as with spreadsheet columns with mixed data), it is easiest to set the format to `Z` so that all columns use their default types.

Importing and Exporting Text Named Data

You can use the `FileExport(_Text, _Named)` format to write the data in variables as columns of ASCII text:

```
FileExport(File, _Named, _Text
           [, Var1 [..., VarN]]
FileImport(File, _Named, _Text
           [, Var1 [..., VarN]][, Start [, Count]])
```

File indicates the file name. *Start* and *Count* can be used to read a section of a file. *Start* indicates the starting row to import, and *Count* indicates the maximum number of rows to read.

`FileImport(_Text, _Named)` expects the file format presented below:

- The first line of the file contains the names of the variables, separated by a special separator character, normally a tab.
- Each line after the first represents a single record of data.

- Scalar values fill only the second row.
- Arrays occupy a number of rows corresponding to the valid range of the array. For instance, an array with elements 5 through 10 has values in rows 6 through 11 of the file.

Although missing entries are left blank, the separator character is still written.

The data determines the variable type. If the data cannot convert to a date-time or real value, it becomes a string. All variables are arrays, even if there is only one element.

You can also import text data from a file in which the data is separated by spaces, instead of the separator character. The variable names occupy the first row. The position of the leftmost character of each variable name within the row determines the position at which data begins for that column, except for the first variable, which begins in column 1. Consider the following file in which a period represents a space:

```
Name.....Salary..Birthdate..SickDays
Asbark Fliegleman.....07/25/68...11
Sandra Mushkin.....36200...11/09/63...7
Jake Drew.....35000...02/21/55
```

CA-Realizer pads each entry with the number of spaces needed to align each value in the column. It fills missing entries entirely with spaces. It imports this data as *Name*[1:3], *Salary*[2:3], *Birthdate*[1:3], and *SickDays*[1:2].

Note: You can change the separator character with the `SetSys(_Separator)` command.

Importing and Exporting Text Plain Data

The `FileImport(_Text, _Plain)` and `FileExport(_Text, _Plain)` commands are similar to their corresponding `_Named` variations. However, the `_Plain` formats allow for greater control over how data is read or written.

```
FileExport(File, _Text, _Plain, Template)
FileImport(File, _Text, _Plain, Template
           [, Start[, Count]])
```

File indicates the file name. *Start* and *Count* can be used to read a section of a file, where *Start* indicates the starting row to import, and *Count* indicates the maximum number of rows to read.

Each column can either be set as a fixed width or separated by the separator character (normally a tab). To set the width of each column (for example 20), specify the field format size with the *FORMAT* member as follows:

```
Template.VARIABLE = {"Name", "Age", "Birthdate"}
Template.FORMAT = {"A20", "R", "D"}
```

If the format size is 0 or unspecified, the separator character is used between columns. When importing data for which the size is specified, CA-Realizer reads and uses exactly that number of bytes. If the size is unspecified, CA-Realizer reads data until the separator character is found.

When exporting numeric and date-time values as fixed width fields with the *Rmm* and *Dmm* formats, *mm* must be at least as large as the default format for the variable. When importing, CA-Realizer reads and converts *mm* characters, if possible, to the format type. If the data cannot be converted, the default values, 0.0 and 01/01/1900, are used for numerics and date-times, respectively.

You can use `SetFormat`, `SetSys(_RealFormat)`, and `SetSys(_DateFormat)` to change the format used to write real and date-time values.

When exporting strings into fixed width fields with the *Amm* format, *mm* must be at least as large as the largest alpha in the column. When importing fixed width fields, *mm* characters are read into the strings and trailing spaces are removed.

If a fixed width field being read is more than *mm* characters wide, excess characters must also be read by skipping the remainder of the characters with an *Xmm* field (where *mm* is the number of characters left unread). *X0* is used to skip until the next separator character.

The *Z* format, without a size, reads in all characters until the next separator character, and determines the type of the column by the entry in the first row. If possible, CA-Realizer converts the value read to a numeric or date-time value. Otherwise, it reads it as a string. If you specify a size, CA-Realizer reads and converts *mm* characters in the same way.

Use the `SetSys(_Separator)` command to change the separator character. For more information, refer the `SetSys` command in the *CA-Realizer Language Reference Guide*.

Binary Files

In a CA-Realizer data file created with `FileExport(_Realizer, _Named)` or `FileExport(_Realizer, _Plain)`, the information is efficiently stored in binary format, which makes it faster to read and write. However, the CA-Realizer file format contains information not readily available to the user, so other applications that want to read or write data in the CA-Realizer format cannot.

The following binary formats enable data to be read and written in a fast and compact binary format that you design. This also allows CA-Realizer to read and write files in the binary formats of any other application program.

Plain Binary Format

Like the other Plain formats, the Binary Plain format requires a template family with *VARIABLE* and *FORMAT* members. The list of formats, however, is more extensive.

```
FileExport(File, _Binary, _Plain, Template)
FileImport(File, _Binary, _Plain, Template
           [, Start [, Count]])
```

As before, specify each format as *fnn*, where *f* is the format character and *nn* is the field width. Unlike other file types, the field width is mandatory for all Binary formats. The following table shows the Binary Plain format types.

Code	Format	nn	Description
A	Alpha (normal)	1...1024	When exported, the string is truncated or null-padded to <i>nn</i> bytes. When imported, trailing nulls are removed.
B	Bytes (pure)	1...1024	Same as A, but trailing nulls are not removed when the data is imported (this can be used for a fixed size buffer of any binary data).
D	Date- Time	4 or 8	D8 contains both date and time information; D4 contains only the date. When importing D4, the time is set to 00:00:00.
I	Integer	1, 2, 3, or 4	Real value converted to a signed integer. (There is no range checking when exporting.)
R	Real	4 or 8	Real value stored as an IEEE floating point value. (There is a possible loss of precision with R4.)
U	Unsigne d Integer	1, 2, 3, or 4	Real value converted to an unsigned integer. (There is no range checking when exporting.)
X	Skip	1...1024	Skip <i>nn</i> bytes of the data record.

The ranges of the numeric types are shown below:

Type	Range
I1	-128 to 127
I2	-32,768 to 32,767
I3	-8,388,608 to 8,388,607
I4	-2,147,483,648 to 2,147,483,647
U1	0 to 255
U2	0 to 65,536
U3	0 to 16,777,215
U4	0 to 4,294,967,295
R4	3.4E-38 to 3.4E+38
R8	1.7E-308 to 1.7E+308

Binary Named Format

The Binary Named format is somewhat different from the other named formats:

```
FileExport(File, _Binary, _Named, Template)
FileImport(File, _Binary, _Named
           [, Var1 [..., VarN]]   [, Start [, Count]])
```

FileExport(_Binary, _Named) requires a template like those used in the Plain formats because the same data can be written in several different formats. FileImport(_Binary, _Named) works the same as the other Named imports.

With Binary Named, the data is stored exactly the same as with Binary Plain, but a header block is written before the data, indicating the names and types of the variables.

The format for the header is shown below:

Name	Type
1 word	The number of fields (<i>F</i>).
1 word	The number of bytes in the variable name list (<i>N</i>).
1 word	The size of one record (sum of <i>W</i> for each field).
1 word	Reserved—must be 0.
<i>N</i> bytes	The list of variable names stored as <i>F</i> null-terminated strings. Empty names may be used for skip fields.
<i>F</i> * 3 bytes	The formats of the fields. Each is a 1-byte character indicating the format, followed by a word containing the width of the field (<i>W</i>).

Note: A word is a 2-byte unsigned integer.

Importing and Exporting XBase Data Files

Import and export XBase files using the FileDB function:

```
family = FileDB(FileName, FileType, Action
[, Start [, Count]] [, Info, Data])
```

FileType is always `_DBF`. *Action* can take the values `_Import`, `_Info`, or `_Export`. *Start* and *Count* specify the starting record number, and number of records to import or export. If you do not specify *Start*, CA-Realizer imports or exports the entire file. *Info* is a family whose members describe the fields in the database. *Data* is the family of data to be imported or exported and it contains a member named after each field listed in the *Info.Fields* array.

XBase files are made up of a number of records of predefined field types. The following table lists these types.

Field Type	Code	Description
Character	C	ASCII character strings
Numeric	N	Numeric values (integer and real) stored in ASCII format
Date	D	Date values stored in ASCII format
Logical	L	Values of either True (T) or False (F)

Each field has an associated length, which represents the maximum number of characters that can be used to store a value in that field:

- Character fields—the length of the space-padded string.
- Numeric fields—total size of the number in ASCII format, including a negative sign or a decimal point (with numeric fields, you can also specify the number of decimal digits that are included as part of the length).
- Date fields—always 8 bytes, stored in *yyyymmdd* format.
- Logical fields—always 1 byte.

Querying a .DBF File

Before you import a XBase file, it is helpful to query the file's header information by specifying the `_Info` action parameter for `FileDB`. You can use this header information later to re-export data from the file.

Query a .DBF file as follows:

```
Info = FileDB(FileName, _DBF, _Info)
```

The following table lists the members of the *Info* family:

Info Member	Returns
<i>Info.NumRecords</i>	Number of records in the file
<i>Info.Fields</i>	Field names
<i>Info.Types</i>	Field types (C, N, L, or M)
<i>Info.Lengths</i>	Field lengths
<i>Info.DecPlaces</i>	Number of decimal places

Try the following on one of the .DBF files in the `REALIZER\MANUAL` directory.

```
FileName = StdOpen("*.DBF")  
Info = FileDB(FileName, _DBF, _Info)  
PRINT Info
```

Importing .DBF Files

CA-Realizer imports XBase files as a family of arrays, where each member is named by the field name in the XBase file.

```
Data = FileDB(FileName, _DBF, _Import  
              [, Start [, Count]])
```

The valid range of the family member arrays represents the record numbers that CA-Realizer imports. If you do not specify *Start*, CA-Realizer imports the entire family. If you specify *Start*, CA-Realizer imports records beginning at *Start*. If you specify *Count*, CA-Realizer imports only that number of records.

The type of family member depends on the type of the field:

XBase	CA-Realizer
Character	String
Numeric	Numeric
Date	Date-time (with time set to 00:00:00)
Logical	Numeric (0 = 'F', 1 = 'T', -1 = '?')

Exporting .DBF Files

You can export an array of families (*Data*) as a XBase file using:

```
FileDB(FileName, _DBF, _Export
      [, Start [, Count]] [, Info, Data] )
```

The valid range of the family member arrays represents the record numbers that CA-Realizer exports. If you do not specify *Start*, CA-Realizer exports the entire family. If you specify *Start*, CA-Realizer exports array elements beginning at *Start*. If you specify *Count*, CA-Realizer exports only that number of elements.

The *Info* family should contain the members listed in the following table with the exception of *NumRecords*:

<i>Info Member</i>	<i>Returns</i>
<i>Info.NumRecords</i>	Number of records in the file
<i>Info.Fields</i>	Field names
<i>Info.Types</i>	Field types (C, N, L, or M)
<i>Info.Lengths</i>	Field lengths
<i>Info.DecPlaces</i>	Number of decimal places

Only *Fields* and *Lengths* are mandatory. *Types* and *DecPlaces* are optional.

The *Data* family must contain a member named after each field listed in the *Info.Fields* array. Each member should be an array of an appropriate type (if the *Info.Types* member was specified).

The following example combines all of the above steps:

- It permits you to select a .DBF file for import.
- It queries the file and displays the contents of *Info* in a spreadsheet.
- It imports the data and displays it in a second spreadsheet.

- It permits you to add new records.
- It exports your record to an XBase file of your choice.

In addition, this example forces you to add new records with the INPUT box. This can be tedious when entering multiple records, and does not allow you to change your mind once a field has been entered. (Later in this guide, examples of forms and spreadsheets that remove these limitations and provide powerful user interfaces, are presented.)

```
' Manual\PG_07\DATABASE.RLZ

'Allow the user to add a record to the database
PROC AddRecord(Info, Data)
    LOCAL NumFields, NumRecs, k
    LOCAL FieldName, Caption, value

    NumFields = EndValid(Info.Fields)
    NumRecs = Info.NumRecords
    FOR k = 1 to NumFields
        'Prompt the user for the field value
        FieldName = Info.Fields[k]
        Caption = "Enter a Value for " + FieldName
        INPUT Caption; value
        'Special case: convert numbers to strings
        IF Info.Types[k] = "C" AND QVar(value, \\  
            _Real) THEN
            value = Sprint(value)
        END IF
        'Update the field
        s = "Data." + FieldName + "[" + \\  
            NumToStr(NumRecs+1) + "]" = value"
        EXECUTE s
    NEXT k
    SheetUpdate(Data)
END PROC

'Ask the user for a database file name
FileName = StdOpen("*.DBF")
IF FileName = "" THEN
    EXIT PROGRAM
END IF

'Create a spreadsheet with the file structure
Info = FileDB(FileName, _DBF, _Info)
SheetNew(1, Info; "Database structure")
SheetControl(_Size; _Center, _Top, 70 pct, 30 pct)
SheetControl(_Show)
```

```
'Create a spreadsheet with the record values
Data = FileDB(FileName, _DBF, _Import)
SheetNew(2, Data; "Database records")
SheetControl(_Size; _Center, 40 pct, 70 pct, 60 pct)
SheetControl(_Show)

LOOP
AddRecord(Info, Data)
IF MessageBox("Add another record", \
  "Database Example", _MB_YesNo) = _No THEN
  EXIT LOOP
END IF
END LOOP

'Export the updated database file
ExportName = StdSaveAs("*.DBF")
IF ExportName <> "" THEN
  FileDB(ExportName, _DBF, _Export; Info, Data)
END IF
```


Chapter 8

Controlling Input and Output

In This Chapter	8-1
The Standard Dialog Boxes	8-1
Selecting a File to Open	8-2
Changing the Printer Configuration	8-6
Changing a Font	8-8
Changing Colors	8-10
The Input Box	8-11
Displaying Messages Using Message Boxes	8-14
Printing	8-16
Printing Arrays	8-17
Printing to Logs	8-17
Formatting Output	8-18
String Formats	8-19
Numeric Formats	8-20
Fractional Formats	8-24
Date-Time Formats	8-27
Mixing Formats	8-27
Printing in Rows	8-28
Changing Default PRINT Formats	8-28
Sending Output to a Printer	8-30

Chapter 8

Controlling Input and Output

In This Chapter

This chapter discusses the use of CA-Realizer's standard dialog box functions for controlling file input and output, including:

- Opening a file
- Saving a file
- Closing a file
- Setting printer options
- Setting fonts
- Setting colors

The INPUT and PRINT commands and MESSAGEBOX function are also discussed.

The Standard Dialog Boxes

CA-Realizer provides several standard dialog box functions to enable your applications to ask the user for file names to use when saving and retrieving files, to input data into your programs, to change colors or fonts, and select printers and printer configurations.

Selecting a File to Open

As discussed in “File I/O”, you can open a file by hard coding the file name when using the FileOpen command. You should, however, be able to design a program with greater flexibility, so that it allows the user the opportunity to choose an appropriate file name.

For example, when you select OPEN from CA-Realizer’s FILE menu, an Open File dialog box appears, displaying a list of file names from which you can choose the file to be opened. When you select SAVE AS from the FILE menu, a Save As dialog box appears—in this dialog box you can type a new file name.

Since choosing file names in this way is common to many applications, CA-Realizer provides functions that generate the Open File and Save As dialog boxes.

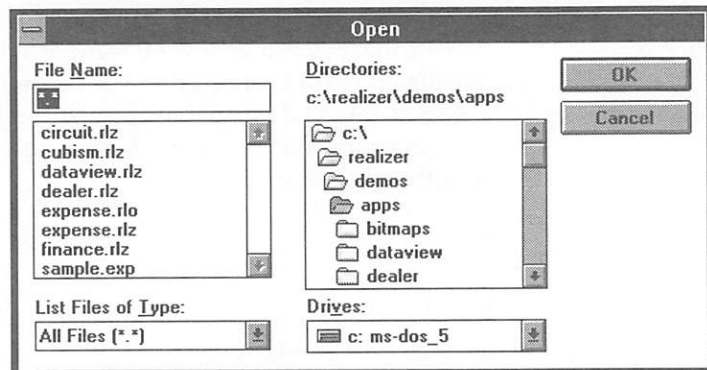
StdOpen

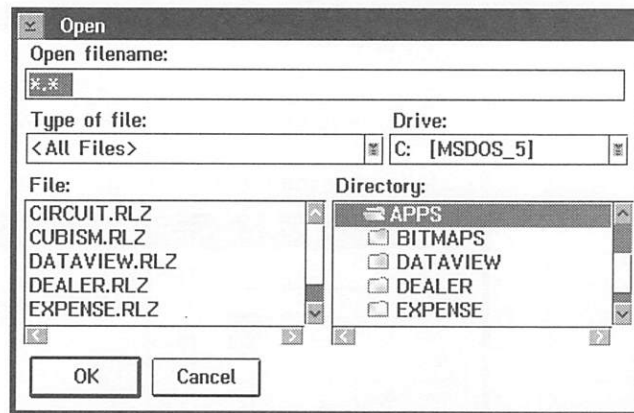
The StdOpen function creates a dialog box that allows the user to select a file for reading or appending data. The user can navigate through directories and specify wild card symbols.

To display the Open File dialog box, use the StdOpen function as follows:

```
FileName = StdOpen(FileFilter [, Caption  
[, OKButtonText [, CancelButtonText]])
```

Both the Windows and OS/2 versions are shown here and throughout this chapter:





FileFilter specifies a file pattern to use as a file filter. (To show all files, use `"*.*`".) To override the default caption of "Open," specify *Caption*. To change the default text within the OK and CANCEL buttons, specify *OKButtonText* and *CancelButtonText*, respectively.

CA-Realizer returns the full path name of the selected file. If the user did not select a file (having pressed either the ESC key or the CANCEL button), the return value is an empty string. If the file name the user entered does not exist, CA-Realizer displays an error message and the Open File dialog box remains on the screen.

For example, the following statement creates an Open File dialog box which contains all files in the current drive and directory with the .RLZ file extension:

```
Filename = StdOpen("*.RLZ")
```

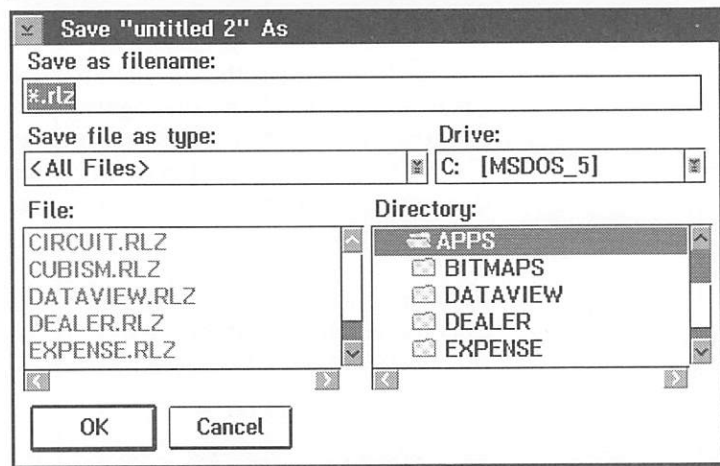
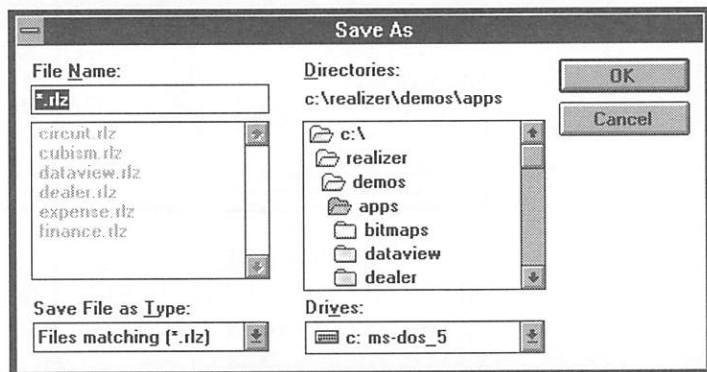
This dialog box can be used to allow the user to select a file and display its contents in a new program window, making the file available for editing.

StdSaveAs

To display the Save As dialog box, use the StdSaveAs function as follows:

```
FileName = StdSaveAs(DefaultFile [, Caption  
[, SaveButtonText [, CancelButtonText ]]])
```

The following dialog box appears:



The Save As dialog box allows the user to specify a file name for the file to be saved.

DefaultFile is the default file name that appears when the dialog box is first displayed. Specify *Caption* to override the default caption of "Save As." To change the text that appears in the OK and CANCEL buttons, specify *SaveButtonText* and *CancelButtonText*, respectively.

If the file name entered by the user is not valid, CA-Realizer displays a message box indicating the file name is invalid. CA-Realizer then returns the user to the Save As dialog box, allowing the file name to be reentered. If the file already exists, CA-Realizer displays a warning message and allows the user to choose a different name.

For example, enter and run the following statements to display the Save As dialog box:

```
Filename = StdSaveAs("SAMPLE.DAT")
PRINT Filename
```

Now type any text into the Save As dialog box. Choose OK when you are finished. Notice that the text you entered appears in the Print Log with the full path name of the current directory.

If the user chooses the CANCEL button or presses ESC, an empty string ("") is returned.

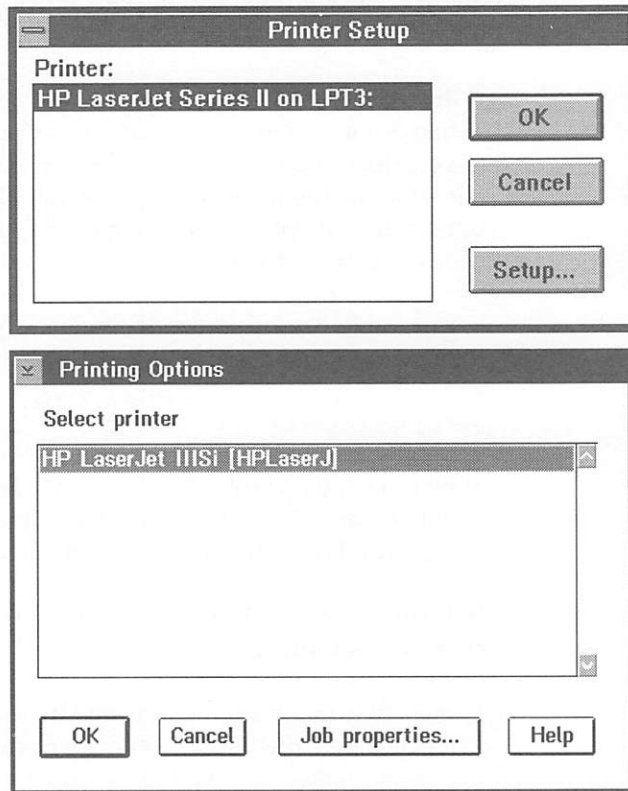
You can use the file name returned by the StdSaveAs function to create a new file with the FileOpen command as follows:

```
FileNum = FileQUnique
Filename = StdSaveAs("SAMPLE.DAT")
IF Filename <> "" THEN
    FileOpen(FileNum, Filename)
END IF
```

Changing the Printer Configuration

StdPrintConfig

The StdPrintConfig command displays the Printer Setup dialog box. This dialog box allows the user to set or modify a variety of printer settings including: fonts, paper orientation, and output.



StdPrintConfig does not return a value. Additionally, the changes made only affect CA-Realizer's printer settings; other applications maintain their own settings.

PrinterControl

The PrinterControl command can also be used to change the printer settings programmatically:

```
PrinterControl (Action; Mod)
```

The following table lists the available *Action* and *Mod* values:

Action	Changes	Mod
_SetMode	Printer Mode	_Portrait or _Landscape
_Copies	Number of copies	NumCopies
_SetHeader	Default page header	Header
_SetFooter	Default page footer	Footer
_SetFont	Default printer font	FontNum

For example, the following statement switches the printer mode to landscape mode:

```
PrinterControl (_SetMode; _Landscape)
```

Changing a Font

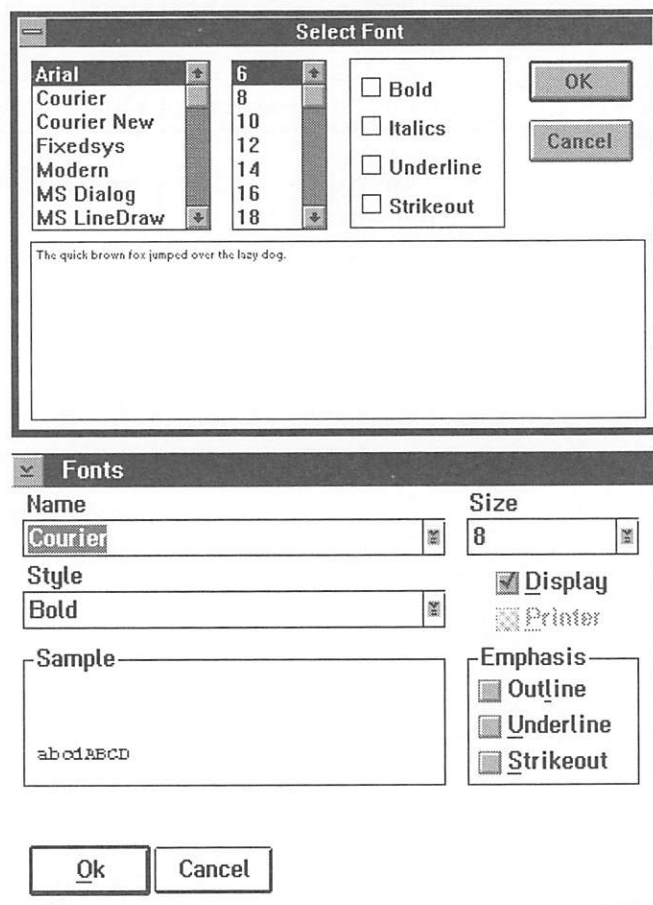
StdFont

The StdFont function creates a dialog box that allows the user to select a font from all the available fonts and sizes, and includes a sample display of the selected font.

To display the Select Font dialog box, use the StdFont function as follows:

```
FontName = StdFont([Caption [, OKButtonText  
[, CancelButtonText [, Sample]]]) [;_Printer])
```

The following dialog box is displayed:



Caption overrides the default caption of “Select Font.” To change the text for the OK and CANCEL buttons, specify *OKButtonText* and *CancelButtonText*, respectively. *Sample* sets the text that you use to demonstrate the selected font.

The *StdFont* function returns a vector containing three elements: *FontName* (for example, *Courier*), *FontSize* (in points), and *FontStyle* (the sum of any of the style values including: *_Bold*, *_Underline*, *_Italics*, and *_Strikeout*).

You can pass this vector directly to the *FontNew* command to use the selected font in a CA-Realizer program. If the user chooses OK or presses ESC, an array of three empty strings is returned.

You can also use *StrToNum* to convert *FontSize* and *FontStyle* to real values. For more information about the *FontNew* and *StdFontConfig* commands, refer to *CA-Realizer Language Reference Guide*.

Note: If the *_Printer* modifier is specified, the font list contains all the available printer fonts instead of the display fonts.

Changing Colors

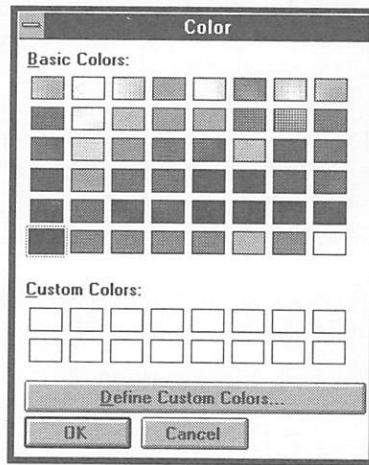
StdColor

StdColor allows the user to select a color. This dialog box displays some of the available colors and a sample of each.

To display the Color dialog box, use the StdColor function:

```
ColorName = StdColor(Initial Color [, Title  
                    [, OKButtonText [, CancelButtonText]])
```

Note that because the Windows and OS/2 StdColor dialog boxes are identical, only the Windows dialog box is presented:



This dialog box enables the user to select a color by choosing one of the predefined CA-Realizer colors or by selecting RGB values with scrolls bars and entry fields.

InitColor can be a numeric value representing one of the predefined CA-Realizer colors, or a three-element vector of RGB values ranging from 0 to 1. The default text for the dialog box window, the OK button, and the CANCEL button can be overridden with *Title*, *OKButtonText*, and *CancelButtonText*, respectively.

StdColor returns a scalar if one of the predefined colors was selected, or a three-element vector of RGB values if the color selected by the user was not predefined. If the user chooses the CANCEL button or presses ESC, a scalar value of 0 is returned.

The Input Box

The INPUT command obtains values from the user. It also provides a way to alert a user when an error has occurred and to pause program execution until the user is ready. The general form of the INPUT command is:

```
INPUT [Prompt [, Title] ;] [Data1 [ ..., DataN]]
```

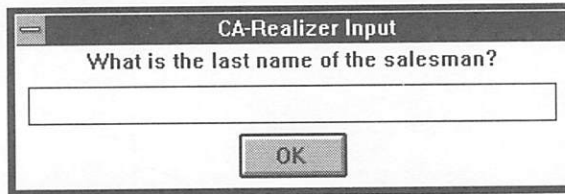
The default prompt is "Expecting *n* values" and the default title is "CA-Realizer Input." Set *Prompt* to override the default prompt and set *Title* to override the default title.

Obtaining Values from the User

The following example creates a variable called *SalesName* and displays the Input dialog box that allows the user to enter a value for *SalesName*:

```
INPUT "What is the last name of the salesman?"; \\  
    SalesName  
PRINT SalesName
```

The CA-Realizer Input dialog box appears as follows:



Note that the *Prompt* string is displayed above an edit field.

Now, enter a salesman's name into the edit field and then choose OK. This stores the input data in the *SalesName* variable.

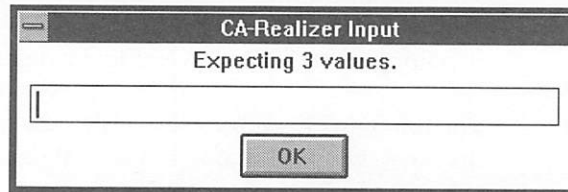
The type of the input variable is automatically determined by the input value. If the input value can be converted to a number, the variable is determined to be numeric. If the input value contains a non-numeric character, the variable is determined to be a string.

The INPUT command can obtain values for more than one variable. If the INPUT command has a prompt string, list input variables after a semicolon. You must separate variable names with commas. Each value can be a different type.

For example, the following command line requests three values:

```
INPUT a, b, c
```

The following INPUT dialog box appears:

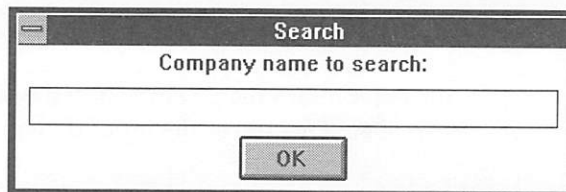


To respond to the dialog box, input the expected variables separated by commas. Note that when you omit a *Prompt* string, CA-Realizer uses its own prompt to indicate how many input values are expected.

The following example changes the default Title to "Search":

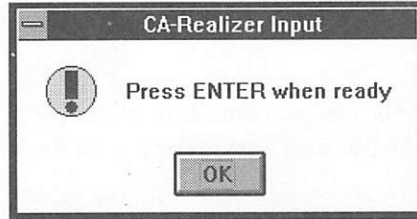
```
INPUT "Company name to search:", "Search"; \\
    SearchName
PRINT SearchName
```

These statements produce the following dialog box:



Using INPUT to Display
Messages

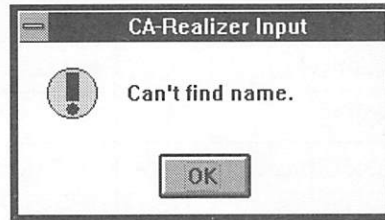
By default, if you omit all arguments from an INPUT command, CA-Realizer displays a warning box as follows:



To override the "Press ENTER when ready" message, specify *Title* only. For example,

```
INPUT "Can't find name.";
```

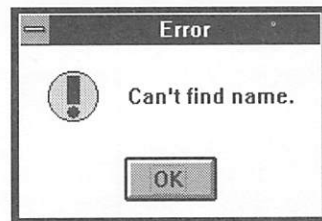
displays the following warning box:



To change the title of the warning box, specify *Title*. For example, the statement,

```
INPUT "Can't find name.", "Error";
```

displays the following Error warning box:



Note that a semicolon must be at the end of an INPUT command which supplies a message.

Displaying Messages Using Message Boxes

You can display more elaborate messages using the `MessageBox` function:

```
ButtonPressed = MessageBox(Text, Caption, Type  
[, Icon [, DefaultButtonNumber]])
```

Caption is a string containing the caption along the top of the message box and *Text* is the text in the message box.

Type determines the type of message box, as well as the number of buttons. It can be any one of the following constants; the following table also lists the values that may be returned for each type of message box:

Constant	Buttons	Return Values
<code>_MB_OK</code>	OK	<code>_OK</code>
<code>_MB_OKCancel</code>	OK, CANCEL	<code>_OK</code> , <code>_Cancel</code>
<code>_MB_YesNO</code>	YES, NO	<code>_Yes</code> , <code>_No</code>
<code>_MB_YesNoCancel</code>	YES, NO, CANCEL	<code>_Yes</code> , <code>_No</code> , <code>_Cancel</code>
<code>_MB_RetryCancel</code>	RETRY, CANCEL	<code>_Retry</code> , <code>_Cancel</code>
<code>_MB_AbortRetryIgnore</code>	ABORT, RETRY, IGNORE	<code>_Abort</code> , <code>_Retry</code> , <code>_Ignore</code>

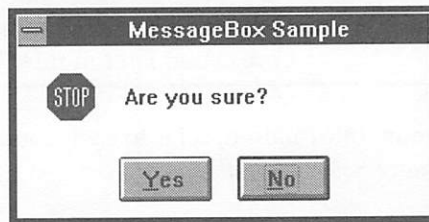
The optional *Icon* parameter specifies an icon to display on the left side of the message box, as listed in the following table:

<i>Icon</i>	<i>Displays</i>
_MB_Stop	
_MB_Exclamation	
_MB_Question	
_MB_Information	

Specify the default button with *DefaultButtonNumber*, which can be 1, 2, or 3, depending on the number of buttons in the message box.

For example, the following statement sets the second button (No) as the default:

```
ButtonPressed = MessageBox("Are you sure?", \\  
    "MessageBox Sample", \\  
    _MB_YesNo , _MB_STOP, 2)
```



Printing

You can control all printing with the PRINT command or one of its three variants: LPRINT, ROWPRINT, or LROWPRINT. The general format for these commands is very similar:

```
[L][ROW]PRINT [# LogNum [, FormNum] ;]  
[USING Format ;] [ Data1 [, ;]  
[ ..., DataN [, ;]]]
```

Command	Description
PRINT	The PRINT command prints the values of variables and expressions to either a log or a printer. If a log is not specified, the output is sent to the Print Log. If the output is being sent to a log in a form, the number of the form must also be specified.
ROWPRINT	ROWPRINT is a variation of the PRINT command that allows you to print arrays in rows instead of columns. The syntax is the same as for PRINT.
LPRINT and LROWPRINT	Use LPRINT and LROWPRINT to send output to the printer. The syntax of these commands is similar to PRINT and ROWPRINT. For more information, see <i>Sending Output to a Printer</i> presented later in this chapter.

For more information, refer to each command in the *CA-Realizer Language Reference Guide*.

When specifying multiple data items, you must separate each pair of items in the data list by a comma or a semicolon. CA-Realizer prints items separated by a comma with a tab between them, while items separated by semicolons are printed adjacent to one another. If the PRINT statement ends with a comma or a semicolon, the carriage return is suppressed.

Printing Arrays

When you pass arrays to PRINT, CA-Realizer arranges them in columns with elements of the same range printed on the same row. CA-Realizer replicates scalars, as shown below, if you have scalars and arrays in the same PRINT statement:

```
A = Index(6)
PRINT A[1:4], "becomes", A[3:6]
1.0000    becomes
2.0000    becomes
3.0000    becomes    3.0000
4.0000    becomes    4.0000
           becomes    5.0000
           becomes    6.0000
```

When you pass families to PRINT, CA-Realizer prints each member in its own column.

Printing to Logs

By default, CA-Realizer displays the output from the PRINT command in the Print Log. You can print to other logs by specifying the log number *LogNum* (if the log is in a form, you must also supply the ID of the form):

For example, the following statement prints to the log whose ID is 2:

```
PRINT # 2; "Hello ID number 2"
```

The following statement prints to log number 1 within form number 3:

```
PRINT # 1, 3; "Hello form number 3"
```

Formatting Output

The USING option of the PRINT command provides for greater flexibility in controlling the appearance of output. It allows you to precisely format each value.

There are many different format codes for strings, date-time values, reals, and fractional formats. This section presents several examples that show how to use format codes; for a complete list and description of these formats, refer to the PRINT command in the *CA-Realizer Language Reference Guide*.

Place the formatting codes within the string *Format*. You can intermingle literal text with these codes. When you use PRINT USING, each value in the data list is substituted for one of the special formats in *Format*, and is formatted accordingly.

For example, the following statement displays a number with two decimal places:

```
PRINT USING "###.##"; 123.4567
123.46
```

In this example, the following statement prints two numbers in a currency format:

```
PRINT USING "$$#####.##"; 725, 1567
$725.00      $1567.00
```

The following statement displays a number with scientific notation:

```
PRINT USING "##.##^"; 12345678
1.23E+7
```

This statement displays two numbers rounded to an integer and separated by colons:

```
PRINT USING "P(0) : "; 123.456, 987.65
123 : 988 :
```

In this example, numbers are displayed as fractions (denominator of 4):

```
PRINT USING "F4(W1 N1/D1), "; -1.25, 3.5, 12
-1 1/4, 3 2/4, 12 0/4,
```

The following statement displays a full date:

```
PRINT USING "D(D5 M4 D2, Y2)"; StrToDate("7/25/68")
Thursday July 25, 1968
```

String Formats

You can format strings by using string format codes that allow you to display either a subset of a string or the string in its entirety. Several examples of string format code usage are presented below.

The ampersand (&) format code allows you to replace its occurrence in a string with another text string. For example, the following statement substitutes the text string "Marsha" for the & character in the string "Hello &, How are you?":

```
PRINT USING "Hello &. How are you?"; "Marsha"
Hello Marsha. How are you?
```

The next example uses the exclamation point (!) format code to include only the first character of the specified string "Howard":

```
PRINT USING "Hello !, how are you?"; "Howard"
Hello H, how are you?
```

To print the first n characters of a string, place any $n-2$ characters between two backslashes as shown below:

```
PRINT USING "Hello, how is ...\"; "Asbark"
Hello, how is Asbar
```

To use a formatting character—such as an ampersand (&)—in the text, place an underscore before it:

```
PRINT USING "How are & _& her brother &"; \\  
"Julie", "Rick"  
How are Julie & her brother Rick
```

If there are more values than formats in *Format*, the formats are repeated from the beginning of the format string. If there are fewer data values than format commands, PRINT stops after it runs out of data as shown below:

```
PRINT USING "Hello &. How are you? "; "Michael", "\\  
"Rachel"  
Hello Michael. How are you? Hello Rachel. How are  
you?  
  
PRINT USING "Greetings &. What & is it?"; \  
"Earthling"  
Greetings Earthling. What
```

Numeric Formats

CA-Realizer provides you with a set of numeric formatting codes that allow you to indicate the number of digits to display and whether to include dollar signs (\$) and asterisks (*).

The number of digits is represented by the pound sign (#). For example, the value 123.4567 would be displayed as 123.46 if you used a format code of ###.##.

The number of leading and trailing decimal places is controlled by the number of #'s. If there are too few trailing #'s, the number is rounded. If there are too many trailing #'s, 0's are added at the end.

Note: The number of characters printed for a number is always the same as the number of characters used in the format code, except in the case of P(x). P(x) prints the number, rounded to 10^{xth}. For example, P(2) would display 123.456 as 100, and P(-2) would display this value as 123.46.

Several examples of how to use the numeric formatting codes are presented below.

The following examples demonstrate the use of the pound sign (#) for specifying the number of digits to display.

The following statement prints the first two digits preceding the decimal point for the value 3.14159:

```
PRINT USING "Value: ##"; 3.14159  
Value: 3
```

The next example prints one digit preceding the decimal point, and two digits following the decimal point, for the value 3.14159:

```
PRINT USING "Value: #.##"; 3.14159  
Value: 3.14
```

The following statement prints one digit preceding the decimal point, and six digits following the decimal point, for the value 3.14159. Note that since the specified value does not have six digits following the decimal point, a 0 is inserted:

```
PRINT USING "Value: #.#####"; 3.14159  
Value: 3.141590
```

If the number of leading #'s is too large, spaces are used as padding. If the number of leading #'s is too small, the entire field is filled with asterisks.

For example:

```
PRINT USING "Value: #####.###"; 3.14159  
Value:      3.142  
  
PRINT USING "Value: #.###"; 123.456  
Value: *****
```

Monetary Values

Monetary values are displayed with the dollar sign (\$) to the left of the number. Replacing the first two #'s with \$'s places a dollar sign directly to the left of the number, with no spaces between them. Replacing only the first # with a \$ places the dollar sign in the leftmost position, with padded spaces between the dollar sign and the number.

For example:

```
PRINT USING "Value: $$##.##"; 3.14159
Value:  $3.14

PRINT USING "Value: $###.##"; 3.14159
Value:  $ 3.14
```

Padding with Asterisks

A single asterisk can be placed in the leftmost position by replacing the first # with an asterisk as follows:

```
PRINT USING "Value: *###.###"; 3.14159
Value: * 3.142
```

To pad extra decimal places with asterisks instead of spaces, use two asterisks in place of the first two #'s. To also obtain a dollar sign, replace the first three #'s with either **\$ (dollar sign adjacent to the number) or \$** (padding asterisks placed between the dollar sign and the number).

The #'s after the first three characters can be replaced with asterisks or dollar signs without affecting the output.

Several examples are presented below:

```
PRINT USING "Value: ***#.###"; 3.14159
Value: ***3.142

PRINT USING "Value: ****.****"; 3.14159
Value: ***3.142

PRINT USING "Value: **$##.****"; 3.14159
Value: ***$3.142

PRINT USING "Value: $***#.****"; 3.14159
Value: $***3.142
```

Exponential Notation Numbers can be displayed in exponential notation by adding three, four, or five carets (^) after the # format. The first two ^'s represent the E+ or E-, and the rest represent the number of decimal places for the exponent (one, two, or three).

All of the decimal places to the left of the decimal point are used, so no padding occurs. If the exponent needs more decimal places than given, the exponent field is filled with asterisks.

Several examples are presented below:

```
PRINT USING "Value: #.##^"; 314.1592
Value: 3.14E+2

PRINT USING "Value: ##.##^"; 314.1592
Value: 31.42E+1

PRINT USING "Value: ###.##^"; 314.1592
Value: 314.16E+00

PRINT USING "Value: $##.##^"; 0.00000314159
Value: $31.42E-007

PRINT USING "Value: #.###^"; 12345678900
Value: 1.235***
```

Rounding

The P(x) format enables a number to be printed with a fixed number of decimal places without the number of leading decimal places explicitly specified. The number is rounded to 10^{xth}. To obtain *n* trailing decimal places, use P(-*n*). To round a number so that the last *n* trailing digits are 0, use P(*n*).

For example:

```
PRINT USING "Value: P(0)"; 314.1592
Value: 314

PRINT USING "Value: P(-2)"; 314.1592
Value: 314.16

PRINT USING "Value: P(2)"; 314.1592
Value: 300
```

The P(x) format is different from the # formats. Since there is no padding, dollar signs and asterisks are meaningless. It is also the only case where the size of the format string is not the same size as the number of characters used to display the number. To print asterisks or dollar signs, prefix them with an underscore to override their normal meaning as format characters.

For example:

```
PRINT USING "Value: _$P(-2) "; 314.1592  
Value: $314.16
```

Fractional Formats

All the previous formats print fractional parts of numbers in decimal format. You can also print the fractional part as a numerator and denominator by using one of the fractional formats.

The general format for fractions is Fdd(xxx), where dd is a number that defines the denominator and is less than or equal to 10000.

xxx, which provides formatting information, is a combination of characters to be printed literally and any of the following fractional format codes: W1, W2, W3, N1, N2, N3, N4, D1, D2, and _ (underscore).

If several format codes occur within a fractional format, they apply to single values in the PRINT list. Any characters can be mixed among the format codes.

Using W2 instead of W1 prevents the leading 0 from being displayed when the whole portion is 0. All three parts of the number need not be used.

Several examples are presented below:

```
PRINT USING "F4(W1 N1/D1). "; 0, 0.25, 2, -3.75, \\  
11.675  
0 0/4. 0 1/4. 2 0/4. -3 3/4. 11 3/4.  
PRINT USING "F4(W2 N1/D1). "; 0, 0.25, 1, 2.5, \\  
-3.75  
0/4. 1/4. 1 0/4. 2 2/4. -3 3/4.  
PRINT USING "F4(W2 and N1 fourths). "; 0, 0.5, 1, \\  
-3.75  
0 fourths. 2 fourths. 1 and 0 fourths. -3 and 3  
fourths.  
PRINT USING "F8(W1-N1). "; 1, 2.5, -3.75, 0.25, \\  
11.675  
1-0. 2-4. -3-6. 0-2. 11-5.
```

W2 prevents the whole portion from being displayed when it is 0 by skipping to the next formatting command. W3 prevents the fractional part from being displayed if the numerator is 0 by skipping everything else in the parentheses.

For example:

```
PRINT USING "F5(W2 and N1 fifths). "; 0, 0.8, 3, 4.4  
0 fifths. 4 fifths. 3 and 0 fifths. 4 and 2 fifths.  
PRINT USING "F5(W3 and N1 fifths). "; 0, 0.8, 3, 4.4  
0. 4 fifths. 3. 4 and 2 fifths.
```

N2 and D2 allow the number to be displayed in its reduced form. Displaying a reduced numerator without its denominator can be misleading.

For example:

```
PRINT USING "F8(W1 N1/D1). "; 2.875, 6.75, -14.5  
2 7/8. 6 6/8. -14 4/8.  
PRINT USING "F8(W1 N2/D2). "; 2.875, 6.75, -14.5  
2 7/8. 6 3/4. -14 1/2.  
PRINT USING "F8(W1-N2/D2). "; 3.125, 3.25, 3.5  
3-1/8. 3-1/4. 3-1/2.  
PRINT USING "F8(W1-N2). "; 3.125, 3.25, 3.5  
3-1. 3-1. 3-1.
```

N3 displays the numerator with a fixed number of digits. Zeros are padded until the numerator is the same size as the denominator - 1 (the largest value for the numerator).

For example:

```
PRINT USING "F100(W1:N3). "; 0.07, 0.59, 1.99
0:07. 0:59. 1:99.
```

```
PRINT USING "F101(W1:N3). "; 7/101, 99/101, 100/101
0:007. 0:099. 0:100.
```

N4 never displays anything. If the numerator is 0, it causes everything else within the parentheses to be ignored. This can be used to prevent fractional parts from being displayed if the fraction is 0.

For example:

```
PRINT USING "F5(W1 wholeN4 and N1 fifths). "; \\  
2.4, 0.8, 3.0  
2 whole and 2 fifths. 0 whole and 4 fifths. 3 whole.  
PRINT USING "F5(W3 whole and N1 fifths). "; 2.4, \\  
0.8, 3.0  
2 whole and 2 fifths. 4 fifths. 3.
```

Note how N4 differs from W3, which prevents the word "whole" from being displayed. If the W, N, or D characters are needed in the format, an underscore should be placed before them as shown below:

```
PRINT USING "F7(_Whole: W1, _Numerator: N1)"; 8.54  
Whole: 8, Numerator: 4
```

Date-Time Formats

Several formatting commands apply to date-time values. The commands are similar in style to the fractional formats, and, like the fractional formats, the size of the printed value is not necessarily the same as the formatting string.

The general date-time format is `D(xxx)`, where `xxx` is a combination of ASCII characters to be printed literally and any of the date-time format codes.

Several examples of how you can use the date-time formats are presented below:

```
dt = StrToDate("4/8/93  2:34 PM")
PRINT dt
04/08/93

PRINT USING "D(Year Y1)";dt
Year 93

PRINT USING "The date is D(M4 D3, Y2)"; dt
The date is April 8th, 1993

PRINT USING "The current time is D(h4:m1 h5)"; dt
The current time is 2:34 PM

PRINT USING "D(D5 M3-D1 h1:m1)"; dt
Monday Apr-08 14:34
```

Mixing Formats

You can mix together formats for strings, numerics, fractions, and date-times, as demonstrated below:

```
PRINT USING "Hey &, it's D(M4 D2). You owe me \\  
$###.##"; "Dave", QDate, 185.98
Hey Dave, it's April 30. You owe me $185.98

Days = Index(4)
Dates = QDate + Days
Principal = 185.98
Rate = 1.0 + 0.15/365
PRINT USING "Hey &, it's D(M4 D2). You owe me \\  
$###.##"; "Dave", Dates, Principal * Rate ^ Days
Hey Dave, it's December 19. You owe me $186.06
Hey Dave, it's December 20. You owe me $186.13
Hey Dave, it's December 21. You owe me $186.21
Hey Dave, it's December 22. You owe me $186.29
```

Printing in Rows

Here is an example with the ROWPRINT command as a variant of the PRINT command. Instead of printing arrays in columns, it prints them in rows:

```
ROWPRINT Index(3), Sin(Index(3))
1 2 3 0.8415 0.9093 0.1411
```

When you use the USING option, arrays are exhausted for each format as the format is encountered:

```
ROWPRINT USING " It is # Hi &. $***"; Index(3), \\
"Hope", Index(3)
It is 1 It is 2 It is 3 Hi Hope. $**1. $**2. $**3
```

Changing Default PRINT Formats

By default, the PRINT command—without a USING format string—displays real numbers with four decimal places, strings in their entirety, and dates as MM/DD/YY.

You can change these default formats by using the SetSys command with the _RealFormat, _AlphaFormat, and _DateFormat constants:

```
SetSys(_RealFormat, NewFormat)
SetSys(_AlphaFormat, NewFormat)
SetSys(_DateFormat, NewFormat)
```

NewFormat can be any format string that is appropriate for the type. These formats work the same as they do with PRINT USING.

The following example displays the specified values in the default formats:

```
PRINT 1234.5678
PRINT "Good Morning"
PRINT QDate
1234.5678
Good Morning
02/27/93
```

The next example changes the default formats:

```
SetSys(_RealFormat, "#####.##")
SetSys(_AlphaFormat, "\.....\")
SetSys(_DateFormat, "D(D5 the D3)")
PRINT 1234.5678
PRINT "Good Morning"
PRINT QDate
```

```
1234.57
Good Mo
Saturday the 27th
```

The default formats are reset with the RESET _SysVar command.

You can also specify a default format for a particular variable using the SetFormat command:

```
SetFormat(Variable [, NewFormat])
```

PRINT statements and the spreadsheet tool use this format every time this variable is displayed. You can override a variable's default format with PRINT USING and other formatting commands.

NewFormat should be a formatting string appropriate to the variable's type. For example, to force CA-Realizer to print a numeric variable with a leading dollar sign, use the following commands:

```
v = Index(3)
SetFormat(v, "$#####.##")
```

Any subsequent PRINT commands use this format:

```
PRINT v
$    1.00
$    2.00
$    3.00
```

If you omit *NewFormat* from the SetFormat command, the variable again uses the default for its type.

Note: QFormat can be used to return the current default format string for a particular variable at any time:

```
DefaultFormatString = QFormat(Var)
```

Sending Output to a Printer

Two commands are available for sending output to a printer—LPRINT and LROWPRINT. The syntax of these commands is identical to PRINT and ROWPRINT.

CA-Realizer collects output generated by the LPRINT and LROWPRINT commands in a temporary output file called a *print file* (this file holds data sent from these commands, as well as data sent by the *TOOLControl(Print)* commands). The print file continues to collect output a page at a time until the LFLUSH command closes it. When LFLUSH completes, the print file is empty.

Like all Windows and OS/2 applications, CA-Realizer sends a print file to the Print Manager, which then places this file in a print queue. This schedules printing tasks from CA-Realizer and other applications in the order in which they are received and prevents the output of one task from mixing with the output of others.

When you print pages by selecting the PRINT command from the FILE menu, CA-Realizer sends them to the Print Manager directly. In this case, there is no need to use LFLUSH.

Chapter 9

Overview of the CA-Realizer Programmable Application Tools

In This Chapter	9-1
Creating Tools	9-2
Assigning Tool IDs	9-4
Generating a Number	9-4
Specifying a Number	9-5
Creating a Tool	9-6
Stand-alone Tools	9-6
Form-Based Tools	9-9
Selecting and Querying a Tool	9-10
Controlling a Tool	9-12
Displaying a Tool	9-13
Sizing and Positioning a Tool	9-14
Available Units	9-15
Available Justification Settings	9-16
Setting a Position and Size	9-17
Limiting a Window's Size	9-18
Converting Between Units	9-18
Printing a Tool	9-19
Closing a Tool	9-20
Setting Tool Procedures	9-21
Specifying a Tool Procedure	9-21
Assigning a Procedure	9-22
Extracting Data from the Passed Parameter Array	9-23
Determining What Invoked a Tool Procedure	9-26
Setting an Alternative Action	9-29

Writing Asynchronous Tool Procedures	9-30
The Application Procedure	9-32
A Tool Example	9-33
Tool Command Summary	9-35
Animator Commands and Functions	9-35
Board Commands and Functions	9-35
Chart Commands and Functions	9-35
Form Commands and Functions	9-36
Graphics Tablet Commands and Functions	9-37
Log Commands and Functions	9-37
Menu Commands and Functions	9-38
Scheduler Commands and Functions	9-38
Spreadsheet Commands and Functions	9-38

Chapter 9

Overview of the CA-Realizer Programmable Application Tools

In This Chapter

This chapter introduces you to CA-Realizer's *programmable application tools* (also simply referred to as *tools*). These tools allow you to integrate advanced features into your programs with a minimum of time and effort, and include charts, spreadsheets, boardwatches, text logs, graphics tablets, animations, menus, and the Scheduler.

Note: Very often, spreadsheets are referred to as *sheets*, boardwatches as *boards*, text logs as *logs*, and graphics tablets as *tablets*.

In this chapter, you will learn how to:

- Create a tool
- Select and query a tool
- Control a tool's size, location, and state
- Print a tool
- Specify tool procedures

Programmable application tools are both easy to use and to integrate within a program. This chapter presents the basic tasks involved with creating and using tools—once you understand the general concepts, you can master each new tool with minimal effort.

Note that many of CA-Realizer's tools share similar commands (for instance, there is a `ChartNew`, `SheetNew`, and `FormNew` command). When describing common behavior among similar tools, this chapter uses a generic command name—like `TOOLNew` or `TOOLSelect`—rather than tool-specific commands, like `ChartNew` or `SheetSelect`. (Keep in mind, however, that there are no actual commands that begin with “*TOOL*”.)

For detailed information about each tool-specific command, you should read the chapter specific to each tool (such as “Charts,” “Spreadsheets,” or “Text Logs”).

Creating Tools

When creating tools in CA-Realizer, you can create them as *stand-alone* tools, which can exist as separate windows, independent of any form, or as *form-based* tools, which can only be created as objects in a form.

The menus and Scheduler tools you can create, however, do not belong to either the stand-alone or the form-based group. They have an existence *totally independent* of a form—menus use commands similar to the other tools but appear in the system menu bar, while there is only one Scheduler, which does not need to be created explicitly.

Note: See the “Custom Menus and Accelerator Keys” and “The Scheduler” chapters in this guide for more information about creating menu and Scheduler tools.

Stand-alone Tools

Stand-alone tools are probably the most flexible and utilized of all available programmable application tools. These tools are displayed in their own window and can be created and manipulated independently of any form. Spreadsheets, charts, text logs, and boardwatches can be created as stand-alone tools.

The following table lists the commands and functions common to each stand-alone tool:

Command	Description
<code>TOOLControl</code>	Allows you to control the current tool.
<code>TOOLNew</code>	Creates a new tool.
<code>TOOLQ</code>	Returns information about the current tool.
<code>TOOLQUnique</code>	Returns a unique tool number.
<code>TOOLSelect</code>	Selects a tool to be the current tool.
<code>TOOLSetProc</code>	Assigns a procedure to the current tool.

Form-based Tools

Tools that are *form-based* differ from stand-alone tools in that they can exist *only* as form objects (that is, not as independent windows). Animation and graphics tablets are form-based. In addition, spreadsheets, charts, text logs, and boardwatch stand-alone tools, can also be created as form-based tools.

When created as form objects, tools are treated like standard interface controls (for example, check boxes and push buttons). (When placed on a form, both tools and interface controls are referred to as *form objects*.)

Note that form-based tools share the same `TOOLControl` and `TOOLSelect` commands as the stand-alone tools, but because of their specialized nature, have little else in common with the first group.

Important! *This chapter—and the next few chapters—focus on creating and manipulating stand-alone tools. For information about creating and controlling form-based tools, see the “Forms” chapter later in this guide.*

Assigning Tool IDs

When creating stand-alone tools, each instance of a tool must have an *identification number* (ID) that is unique for that type of tool. For example, you can have a Chart 10, a Form 10, a Sheet 20, and a Log 20 all at the same time. Tool ID numbers can range from 1 to 32767.

A tool's ID is assigned to it when it is created and cannot change during the execution of a program. The ID is the mechanism by which you refer to a tool throughout your program. You must, therefore, know the tool's ID or the variable name to which it has been assigned.

To assign a tool ID, you can instruct CA-Realizer to automatically generate a unique number for you, or you can specify a number yourself.

Generating a Number

To generate a unique ID for a new tool, use a *TOOLQUnique* function (such as *ChartQUnique* or *LogQUnique*). For example, the following statements generate an ID for a new chart (assigning it to the *MyChart* variable) and then use that number to create the chart:

```
MyChart = ChartQUnique  
ChartNew(MyChart)
```

You can then use the ID to manipulate the chart (for example, to select or close it).

Note that the lifetime of the ID assigned to *MyChart* is only for the duration of the execution of the code; it is not a permanently fixed value. Therefore, when generated by a *TOOLQUnique* function, the ID of a tool may be different each time the program is run.

Specifying a Number

You can also set the ID of a new tool yourself when creating the tool. Simply supply the desired value when executing a *TOOLNew* function (*TOOLNew* then returns the ID of the tool that was created). For example:

```
ChartNew(10)
```

You must then use that fixed value to reference the tool in the rest of the program. Unlike IDs generated by CA-Realizer, explicitly specified IDs are the same from session to session.

Note: If you specify an ID for a new tool that is already used by an existing tool of that type, the old tool is destroyed and the specified ID is set for the new tool. For example, creating a Chart 10 destroys any previous Chart 10, but does not affect a Form 10 or a Sheet 10.

Note that executing a *TOOLNew* function in certain ways will generate unique IDs. If you omit the tool ID, or explicitly set it to 0, a unique ID is set for the new tool. For example:

```
MyChart = ChartNew(0)
```

```
MyChart = ChartNew
```

In both cases, *ChartNew* creates a unique ID for the new chart tool, and assigns the value to the variable "MyChart."

Tip: It is recommended that you avoid mixing ID assignment methods. To generate a tool ID, either explicitly specify the tool ID yourself, or use the *TOOLQUnique* or *TOOLNew* function to generate a unique number and assign it to a variable.

Creating a Tool

As previously mentioned, you can create a tool as either a stand-alone tool or as a form-based object.

Stand-alone Tools

To create a stand-alone tool that will exist as an independent window, use a *TOOLNew* command (such as *ChartNew* or *SheetNew*). The general form of *TOOLNew* is as follows:

```
TOOLNew ([ToolNumber] [; Title [, Style]])
```

ToolNumber is the ID to be set for the tool (see Assigning Tool IDs for details). A tool's title, by default, is *Tool n*, where *Tool* is the name of the tool type, and *n* is the tool ID. (For example, a form with an ID of 20 has "Form 20" as its title.) If desired, use the optional *Title* parameter to change the default title.

The default style of the tool's window depends on the type of the tool (each tool's default style is discussed in its specific chapter). To set a specific style for a tool when you create it (thereby overriding the default), specify the *Style* parameter as a sum of any of the following constants:

Constant	Description
<code>_Borderless</code>	Creates a tool window without any border at all. <code>_Borderless</code> cannot be used with <code>_Title</code> , <code>_Frame</code> , or <code>_Size</code> .
<code>_Close</code>	Allows a tool window to be closed from the CA-Realizer FILE menu as well as from the tool window's system menu.
<code>_Frame</code>	Places a frame around a tool's window. <code>_Frame</code> cannot be used with <code>_Title</code> or <code>_Size</code> .

Continued

Continued

Constant	Description
<code>_HotClick</code>	Processes a mouse click in a tool window, even if the window is not active. Without <code>_HotClick</code> , clicking a tool window when it is not the topmost window brings it forward <i>without</i> processing the click (i.e., clicking a button in a form will bring the form window forward without invoking the button). With <code>_HotClick</code> , the tool window is brought forward <i>and</i> the mouse click is processed.
<code>_Minimize</code>	Allows a tool window to be minimized (from the system menu in the tool window and with the minimize icon if you also use <code>_Title</code>).
<code>_NoMax</code>	Prevents a tool window from being maximized. The <code>MAXIMIZE</code> command will not be available from the system menu, and if you also use <code>_Title</code> to add a title bar to the tool window, it will not contain a maximize icon.
<code>_Size</code>	Places a resizable border around a tool window, allowing the window to be resized. If <code>_Title</code> is also set, a maximize icon appears in the title bar.
<code>_Title</code>	Adds a title bar to a tool window, which includes a system menu icon, a maximize icon (if you also use <code>_Size</code>), and a minimize icon (if you also use <code>_Minimize</code>). A title bar enables the user to move the tool window.

Note: If the *Style* parameter is set to 0, none of the default styles are used and the tool is displayed with just a thin border.

For example, the default style for a form is `_Frame + _Minimize + _Close`. This puts a frame around the form window, allows it to be minimized, and permits it to be closed from both the FILE menu and the system menu.

However, since there is no title bar, no system menu is displayed. Therefore, the only way to minimize the form window is by pressing `CTRL+F9`. Likewise, you can only close the form window by selecting the `CLOSE` command from the CA-Realizer FILE menu (or by using its keyboard equivalent `CTRL+F4`).

To override the form's default style and add a title bar, specify the following statement:

```
FormNew(10; "My Form", _Title + _Minimize + _Close)
```

This creates a form with an ID of 10, a title of "My Form", and a title bar. In addition, you will still be able to minimize and close the form window, except now you can use the minimize and system menu icons in the title bar.

Tip: When you create a tool, it is automatically assigned a default size and position, depending on the type of the tool and the number of other tools that are already displayed. See *Sizing and Positioning a Tool* for more information about adjusting a tool's default size and position.

Form-Based Tools

Along with the standard interface objects, such as edit controls and push buttons, you can place all of CA-Realizer's tools in forms, with the exception of menus and the Scheduler. Tools in forms operate in almost the same way as stand-alone tools, with a few minor differences:

- Form-based tools are created using a `FormSetObject` command, not a `TOOLNew` command.
- The ID of a form-based tool is its *object number* (that is, the number that was assigned to it when it was created in the form). You cannot use a `TOOLQUnique` function to generate this number.
- When a form-based tool is created, it becomes the current tool (just as if it had been created with a `TOOLNew` command).
- When specifying a form-based tool for use in a `TOOLSelect` or `TOOLQ` command, you must specify its object number, as well as the form number of the form in which the tool is located.

Note: For information about creating and controlling form-based tools, see the "Forms" chapter later in this guide.

Selecting and Querying a Tool

Selecting a Tool

When working with stand-alone tools, most of the commands that you can use with them affect the *current tool* (you therefore do not need to specify a tool ID).

A tool becomes the current tool when you *create* or *select* it. Note that at any given time, you can have one current tool per tool type (for example, a current chart, a current sheet, and a current board).

When you create a tool using a *TOOLNew* command (such as *ChartNew* or *BoardNew*), the new tool becomes the current tool *of that type*. (See *Creating a Tool* for details about *TOOLNew*.)

You can also select an existing tool to be the current tool (of that type) with a *TOOLSelect* command (such as *LogSelect* or *BoardSelect*). The general form is as follows,

```
TOOLSelect (TOOLNumber [, FormNumber])
```

where *TOOLNumber* is the ID of the tool to be selected. (If the tool is embedded in a form, you must also supply the unique number of the form in which it is located.)

Querying a Tool

To get information about a tool, use a *TOOLQ* function (such as *ChartQ*). The general form of these functions is:

```
Info = TOOLQ(Query [, TOOLNumber [, FormNumber]])
```

The constant you supply for *Query* determines what type of information a *TOOLQ* function will return (available constants are summarized in the table below).

By default, *TOOLQ* returns information about the currently selected tool of the specified type. However, you can specify another tool's ID using a *TOOLNumber* parameter to get information about that tool.

Note: If there is no current tool of the specified type (or with the specified ID), *TOOLQ* returns 0.

Also, like *TOOLSelect*, if the tool is embedded in a form, you must also supply the unique number of the form in which it is located.

Query can be one of the following constants:

Constant	Description
_Exists	Indicates whether the current or specified tool exists.
_HWnd	Returns the window handle of the current or specified tool (for use with external procedures and custom controls).
_Selected	Returns the ID of the currently selected tool. (<i>ToolNumber</i> and <i>FormNumber</i> may not be specified with this query).
_Size	Returns the current size and position of the current or specified tool.
_Style	Returns the style that was set for the current or specified tool when it was created.
_Title	Returns the window title of the current or specified tool.

A *TOOLQ* function returns either a scalar or a two-element array. For details on the particular values returned for each type of tool, refer to your *CA-Realizer Language Reference Guide*.

Controlling a Tool

After creating a stand-alone tool, you can use a *TOOLControl* command to control it—for example, to show it, set its size and location, print it, and close it.

Note that a *TOOLControl* command affects the current tool of the specified type—for example, the *SheetControl* command acts on the current sheet, while *BoardControl* affects the current board.

The general form of *TOOLControl* is as follows:

```
TOOLControl(Action [; Left, Top [, Width, Height]])
```

Action can be one of the following constants:

Constant	Description
<code>_Close</code>	Closes the current tool window, thereby destroying it. (See <i>Closing a Tool</i> below.)
<code>_Hide</code>	Hides the current tool window, making it invisible.
<code>_Maximize</code>	Enlarges the current tool window to the full size of the CA-Realizer window.
<code>_Minimize</code>	Reduces the current tool window to an icon.
<code>_Print</code>	Prints the current tool window. (See <i>Printing a Tool</i> below.)
<code>_Restore</code>	Returns a minimized or maximized tool window to its former size.

Continued

Continued

Constant	Description
<code>_Show</code>	Makes the current tool window visible, bringing it to the front of all the other currently displayed windows. (See <i>Displaying a Tool</i> below.)
<code>_Size</code>	Moves and/or sizes the current tool window. This constant requires that you specify the <i>Left</i> and <i>Top</i> parameters to indicate location of the upper-left corner of the tool window. You can also optionally specify <i>Width</i> and <i>Height</i> to indicate the desired window size. (See <i>Sizing and Positioning a Tool</i> below for more information about the <code>_Size</code> constant.)

Note: For details about additional tool-specific *Action* constants, refer to the appropriate *TOOLControl* commands in the *CA-Realizer Language Reference Guide*.

Displaying a Tool

When you create a stand-alone tool with a *TOOLNew* command, CA-Realizer does not automatically display it. This allows you to first add data, objects, and other attributes to the tool before it is displayed.

When you are ready to display the tool, execute a *TOOLControl* command with the `_Show` constant. It will display the current tool of the specified tool type. For example, the following displays the current board:

```
BoardControl(_Show)
```

Sizing and Positioning a Tool

When you create a stand-alone tool, it is automatically assigned a default size and position. The default values used depend on both the types of the tool and the number of other tools that are already displayed.

You can set a new location and size for the current tool of a specified type by using the `_Size` constant with a `TOOLControl` command. With `_Size`, you must specify the new location of the window; in addition, you can optionally set a new width and height for the window.

Note: Specifying a *location* sets the position of the upper-left corner of the tool window, relative to the upper-left corner of the client region of the CA-Realizer window.

Positioning and sizing tool windows in CA-Realizer requires that you supply one or more *values* (for example, 1, 10, or 100) to indicate the desired location and size. Additionally, the given values must be expressed in valid *units* (for example, characters, pixels, or twips). The upper-left corner of a window is pixel location (0, 0).

Note: In general, values indicating a horizontal position increase from left to right, and values indicating a vertical position increase from top to bottom.

Available Units

The following table presents all the *position notation* units supported by CA-Realizer; they are listed alphabetically:

Unit	Description
char	Characters of a specific font
in	Inches
line	Lines of a specific font
mm	Millimeters
pct	Percentage of a window
pt	Points (1/72 of an inch)
pxl	Pixels
twip	Twips (1/20 of a point, or 1/1440 of an inch)

For example, you might position and size a chart as follows:

```
ChartControl(_Size; 100 pxl, 100 pxl, 3 in, 2 in)
```

This statement uses a combination of units to place the upper-left corner of the 3x2 inch chart window at (100, 100) (pixels, relative to the CA-Realizer window).

If you specify a value without explicitly stating the unit to be used, CA-Realizer assumes the following:

- Values between 0 and 1 are assumed to be percentages, where 1 is 100 percent
- Values from 1.1 and above are assumed to be pixels

Percentages

When specifying the size and placement of a stand-alone tool, *percentages* are relative to the size of the inside of the CA-Realizer window. In contrast, when you describe the size and position of a form-based object (such as a button in a form), percentages are relative to the size of the inside of the form.

You will find that percentages are very useful when you need to develop applications that will run on monitors of different resolutions. For example, when you specify forms in percentages and develop them on a VGA monitor, they will scale correctly when the application is run on a SuperVGA monitor.

Line and Char

Note that the *line* and *char* units are used only when sizing forms and objects within a form. In this case, *line* and *char* represent the average height and width of a character drawn using the default font of the form. These units are useful for creating forms that are more independent of the resolution of the monitor on which the form is displayed. The default font for a form can be changed with the `FormControl(_SetFont)` command.

Available Justification Settings

When specifying the position of the upper-left corner of a tool window (or an object in a form), you can also use the following special justification constants:

Constant	Description
<code>_Left</code>	Left justified
<code>_Right</code>	Right justified
<code>_Bottom</code>	Bottom justified
<code>_Top</code>	Top justified
<code>_Center</code>	Centered
<code>_Default</code>	Uses default width and height

When used, these constants let you set the justification of a tool window (or object), as well as its location and size. You must specify the units. For example:

```
20 char(_Center)      '20 chars right of center
-5 line(_Bottom)      '5 lines from the bottom
25 pct(_Left)         '25 pct from the left side
-10 pxl(_Right)       '10 pxl from the right side
```

Setting a Position and Size

The Entire Tool
Window

To set the position and size of the current tool window, use the `_Size` constant with a `TOOLControl` command. The general form is as follows:

```
TOOLControl(_Size; Left, Top [, Width, Height])
```

where *Left* and *Top* represent the location of the upper-left corner of the window, and *Width* and *Height* are the optional new width and height of the tool. (All four of these values must be expressed using valid CA-Realizer position notation.)

Note: Remember, the `TOOLControl` command affects the current tool of the specified type.

For example:

```
'Make a form take up the entire screen
FormControl(_Size; 0 pct, 0 pct, 100 pct, 100 pct)

'Center a form, 60% wide and 40% high, relative to
'the screen
FormControl(_Size; _Center, _Center, 60 pct, 40 pct)

'Position a log as 1/4 of the screen in the
'lower-right corner
LogControl(_Size; _Right, _Bottom, 50 pct, 50 pct)

'Create a 4 inch edit control in a form. Center it
'50 pixels from the top, using a default height.
FormSetObject(10, _TextBox, "", _Center, 50 pxl, \
4 in, _Default)
```

The Client Area of the Tool Window

When you specify the size of a tool window using `_Size`, the title bar and the window frame (if specified) are also sized. If you want to size just the *inside* of the window (the client region), use the `_SizeInside` constant with a `TOOLControl` command instead of `_Size`.

When `_SizeInside` is specified, *Left*, *Top*, *Width*, and *Height* refer to the inside of the tool widow.

Limiting a Window's Size

It is sometimes useful to restrict how small or how large a tool window can become. The `TOOLControl(_SizeTrack)` command allows you to specify a minimum and maximum width and height for the window:

```
TOOLControl(_SizeTrack; MinWidth, MinHeight  
[, MaxWidth, MaxHeight])
```

The values for *MinWidth*, *MinHeight*, *MaxWidth*, and *MaxHeight* must be specified in pixels. For example:

```
ChartControl(_SizeTrack; 50, 100, 300, 250)
```

Converting Between Units

Converting between inches, points, twips, and millimeters is easy; all you need is the conversion factor. To convert from pixels to inches use, the `_HPxlPerInch` and `_VPxlPerInch` constants, which are the number of pixels per horizontal and vertical inch of the screen, respectively.

For example:

```
'How many pixels is a 3 inch x 2 inch object  
width = 3 * _HPxlPerInch  
height = 2 * _VPxlPerInch
```

Note: Do not use position notation anywhere except in a command in which position values are required. Additionally, expressions involving positions must not mix units.

For example, the following statements are processed correctly:

```
x = 3
y = 2
BoardControl(_Size; (x + y) in, (x - y) in)
```

while these statements produce incorrect results:

```
x = 3 in
y = 2 in
BoardControl(_Size; x + y, x - y)
BoardControl(_Size; 5 in + 3 mm, 50 pct - 10 px1)
```

Printing a Tool

You can use a *TOOLControl* command with the *_Print* constant to print the current tool window at any size and position on a page:

```
ToolControl(_Print [; [PrintStyles] [, Left, Top
                    [, Width, Height]])
```

PrintStyles controls the appearance of the printed tool, and can be a sum of any of the following constants:

Constant	Does Not Print
<i>_NoHeader</i>	A header at the top of the page.
<i>_NoFooter</i>	The page number at the bottom of the page.
<i>_NoHLines</i>	Horizontal grid lines (boards and sheets).
<i>_NoVLines</i>	Vertical grid lines (boards and sheets).
<i>_NoColHeaders</i>	Column headings (boards and sheets).
<i>_NoRowHeaders</i>	Row headings (boards and sheets).
<i>_NoFrame</i>	Form border (forms).

When *_Print* is specified, *Left*, *Top*, *Width*, and *Height* refer to the position and size of the printed tool, relative to the size of the printed page. (All four of these values must be expressed using valid CA-Realizer position notation.)

For example:

```
SheetControl(_Print; _NoColHeaders+_NoRowHeaders)
```

Using a `TOOLControl(_Print)` command, you can print multiple tools on a single page. You can mix the output with text from `LPRINT` and `LROWPRINT` commands.

Like `LPRINT`, a `TOOLControl(_Print)` command sends data to a temporary output file called the *print file*. This file accumulates output one page at a time until the `LFLUSH` command closes it. Like other Windows and OS/2 applications, CA-Realizer sends the print file to the Print Manager.

Note: Tools printed with the `PRINT` command on the CA-Realizer FILE menu are sent directly to the Print Manager. In this case, you can only print one tool on each page.

Closing a Tool

A stand-alone tool is closed—and thereby destroyed—with a `TOOLControl(_Close)` command. This command closes the current tool of the specified type. For example, the following closes the current chart:

```
ChartControl(_Close)
```

You can also permit a tool to be closed by a user by creating the tool window using the `_Close` window style.

Setting Tool Procedures

A CA-Realizer tool can be customized by attaching a *procedure* to it. When you click in, close, or otherwise manipulate the tool, CA-Realizer invokes its procedure.

Tool procedures differ in purpose for each type of tool. For example:

- Chart procedures can react to a user clicking on a graph by bringing up another plot with more detailed information.
- Sheet procedures can react to double-clicks in a spreadsheet cell and can also respond to every change the user makes.
- Log procedures are often used to prevent the log from being closed without confirming whether the log text should be saved.
- Form procedures are used to process button clicks, option button changes, and other user actions.
- Menu procedures react to the selection of an item in a menu.

In addition, all tools respond to being closed, and can optionally respond when the user resizes the tool's window.

Specifying a Tool Procedure

Use *TOOLSetProc* to assign a procedure to the current tool of the specified tool type (for example, the *SheetSetProc* command will set a procedure for the current sheet).

Assigning a Procedure

In general, *TOOLSetProc* can be specified as:

```
TOOLSetProc(ProcName; [ MessageList])  
TOOLSetProc(ModuleName; [ MessageList])
```

ProcName is the name of a defined procedure; *ModuleName* is a string containing the file name of a module. The specified procedure or module is executed when the user clicks in the tool, tries to close it, or otherwise manipulates it. (If no procedure or macro is specified, any procedure attached to the current tool is disassociated from it.)

Note: *MessageList* represents a list of one or more messages that can be sent to a procedure; the messages that can be sent depend on the type of the tool.

When a tool invokes a tool procedure, CA-Realizer passes a *tool procedure parameter array* to the procedure. This array contains information about the tool that invoked it; every tool procedure you create should expect to receive and handle this array.

One very useful feature is the ability to attach the same procedure to multiple tools, even if they are not of the same type. You can then use the information in the tool procedure parameter array to determine which specific tool was activated.

Note: You can change the procedure set for a specific tool by simply issuing another *TOOLSetProc* command that references the new procedure.

Extracting Data from the Passed Parameter Array

When a tool procedure receives a tool procedure parameter array, it must know how and where to extract information from it. The elements in tool procedure parameter arrays utilize a set of standard constants, which can be used to index the array.

For example, the following log procedure extracts the data from two of the elements (`_ItemNum` and `_FormNum`) in the passed array (*params*) for use in a `LogSelect` command:

```
PROC MyLogProcedure(params)
    LogSelect(params[_ItemNum], params[_FormNum])
    .
    .
    .
END PROC
MyLogProcedure(array)
```

The standard constants used in tool procedure parameter arrays are listed alphabetically below; the table also lists the contents of the array element with which each constant is associated.

Constant	Corresponding Element Contains
<code>_ControlHeld</code>	A Boolean value (0 or 1) indicating whether the CTRL key was held down during the action that invoked the tool procedure. This can be useful to differentiate CTRL+clicks from normal clicks.
<code>_CustomLP</code>	Values used for messages received from custom controls. (Also see Determining What Invoked a Tool Procedure.)
<code>_CustomWP</code>	Values used for messages received from custom controls. (Also see Determining What Invoked a Tool Procedure.)
<code>_DragForm</code>	In a <code>_DragNDrop</code> message, if an object being dragged invoked the tool procedure, this returns the ID of the form from which the object was dragged. (Also see Determining What Invoked a Tool Procedure.)
<code>_DragItem</code>	In a <code>DragNDrop</code> message, if an item being dragged invoked the tool procedure, this returns the ID of the item that was dragged. (Also see Determining What Invoked a Tool Procedure.)
<code>_FormNum</code>	If an object in a form invoked the tool procedure, this returns the ID of the form that contains the object. (Also see Determining What Invoked a Tool Procedure.)
<code>_Invoke</code>	A constant indicating the reason the tool procedure was invoked, such as <code>_Click</code> or <code>_Close</code> . (Also see Determining What Invoked a Tool Procedure.)

continued

continued

Constant	Corresponding Element Contains
<code>_ItemNum</code>	If a tool, object, or menu item invoked the tool procedure, this returns the ID of the item that was selected. (Also see <i>Determining What Invoked a Tool Procedure</i> .)
<code>_ItemType</code>	If a tool, object, or menu item invoked the tool procedure, this returns the type of the item that was selected. (Also see <i>Determining What Invoked a Tool Procedure</i> .)
<code>_MenuNum</code>	If a menu item invoked the tool procedure, this returns the ID of the menu containing that menu item. (Also see <i>Determining What Invoked a Tool Procedure</i> .)
<code>_ShiftHeld</code>	A Boolean value (0 or 1) indicating whether the SHIFT key was held during the action that invoked the tool procedure. This can be useful to differentiate SHIFT+clicks from normal clicks.
<code>_UseRealizer</code>	A value that can be manipulated for special processing. (See <i>Setting an Alternative Action</i> for more information.)
<code>_XPos</code>	If the tool procedure was invoked by a mouse click, <code>_XPos</code> returns the x position of the mouse cursor, in pixels, relative to the upper-left corner of the tool or object within a form. If it occurred in a spreadsheet, this number indicates the column number of the edited or double-clicked cell.
<code>_YPos</code>	If the tool procedure was invoked by a mouse click, <code>_YPos</code> returns the y position of the mouse cursor, in pixels, relative to the upper-left corner of the tool or object within a form. If it occurred in a spreadsheet, this number indicates the row number of the edited or double-clicked cell.

Note: Modules receive a tool procedure parameter array as its first parameter; the array can then be referenced by __p1 (two underscores) or QOptParam(1).

Determining What Invoked a Tool Procedure

The Action

Typically, a tool procedure should contain an IF or SELECT statement at the beginning that determines what action occurred to invoke the procedure. The _Invoke element in the tool procedure parameter array contains a constant that indicates why the procedure was invoked.

For example, the following form procedure uses the value in the _Invoke element to determine which action to take:

```
PROC MyFormProcedure(params)
  FormSelect(params[_FormNum])
  SELECT CASE params[_Invoke]
    CASE _Close
      'Process _Close message
    CASE _Click
      'Process _Click message
    CASE _Size
      'Process _Size message
  END SELECT
END PROC
```

The constants that can be found in the `_Invoke` element are listed below in alphabetical order; the table also indicates what user action generates each constant:

Constant	Description
<code>_Change</code>	Generated when the user either changes the value of a cell in a spreadsheet (optional), or selects a new object in a list box that has the <code>_Notify</code> style flag (always generated).
<code>_Click</code>	Generated based on various actions, depending on the tool (always generated).
<code>_Close</code>	Generated when the user tries to close a tool (always generated).
<code>_Custom</code>	Generated by custom controls (always generated).
<code>_CustomNow</code>	Generated by custom controls (always generated).
<code>_DragNDrop</code>	Generated when a draggable object is moved onto a drag-receivable object or form (always generated—forms only).
<code>_GetFocus</code>	Generated when a tool window becomes the topmost window, or when an object in a form gets input focus (optional).
<code>_MiddleClick</code>	Generated when an object in a form is clicked with the middle mouse button (optional—forms only).
<code>_RightClick</code>	Generated when an object in a form is clicked with the right mouse button (optional—forms only).
<code>_Size</code>	Generated when the tool window is resized (optional).

The Item (Tool, Object, or Menu Item)

You can also use the data in a tool procedure parameter array to discover what tool, form object, or menu item invoked a tool procedure. The `_FormNum`, `_ItemNum`, `_ItemType`, `_MenuNum`, `_DragForm`, `_DragItem`, `_CustomWP`, and `_CustomLP` elements in the passed parameter array can be used to find out this information.

`_FormNum`. Contains the ID of the form that was closed or in which an object was selected.

`_ItemNum`. Contains the ID of the item (tool window, form object, or menu item) that was selected (thereby invoking the tool procedure).

Note that there are also three special values for `params[_ItemNum]`:

Value	Description
-1	No specific item has been selected (for example, tool is being closed).
1	The user pressed ENTER in a form.
2	The user pressed ESCAPE in a form

`_ItemType`. Contains a constant (`_Chart`, `_Sheet`, `_Board`, `_Log`, `_Form`, or `_Menu`) that indicates the type of tool that invoked the procedure. This is useful in the help procedure or if you use one procedure to manage tools of different types.

`_MenuNum`. Contains the ID of the menu containing the menu item that invoked the tool procedure (if applicable).

_DragForm and _DragItem. When a form receives `_DragNDrop` messages, the `_ItemNum` and `_FormNum` elements in the parameter array contain the IDs of the object and the form that *received* the object (for example, the object on which the dragged item was dropped), respectively. Additionally, the `_DragItem` and `_DragForm` elements indicate the ID of the object that was dragged and the form *from which* it was dragged. (Drag and drop is discussed in the “Forms” chapter.)

_CustomWP and _CustomLP. When a form receives a message from a custom control object, the tool procedure for the form containing the object is invoked with a `_Custom` or `_CustomNow` message. When this occurs, the `_CustomWP` and `_CustomLP` elements contain the `wParam` and `lParam` or `mParam1` and `mParam2` values from the message.

Setting an Alternative Action

You can change the value of the `_UseRealizer` element in a tool procedure parameter array to force CA-Realizer to take a different action than it normally would have. For example, when a tool procedure is called because a tool is being closed, `_UseRealizer` is set to 1. If you change the value to 0, the tool will not be closed.

If the procedure is a help procedure, it was called because the user pressed the F1 key. In this case, CA-Realizer’s normal help would not be displayed, and `_UseRealizer` is set to 0. If it is changed to 1, CA-Realizer’s normal help will be displayed.

Writing Asynchronous Tool Procedures

A tool procedure is invoked when a user clicks in or tries to close a tool. These actions can come at any time and in any order, so you must prepare a program to deal with them *asynchronously*.

A common method for handling asynchronous tool processing is to:

1. Create a tool.
2. Generate any data for it.
3. Assign a procedure to it.
4. Let the program end.

CA-Realizer is not running any part of the program, but the user can manipulate the windows and cause actions to happen automatically. CA-Realizer waits for an event to occur before routing processing to the appropriate tool procedure.

For instance, if a chart is resized, CA-Realizer rescales the data by itself. If a spreadsheet is scrolled, the data within the sheet is updated accordingly. Likewise, if a user clicks or tries to close a tool, it invokes the procedure for that tool.

Once a tool procedure does what it needs to (for example, updating data or creating new tools), the procedure ends and CA-Realizer continues waiting for additional events to happen.

Tool procedures, therefore, must be designed so that they perform their processing asynchronously. They also should not make any assumptions about what the current tool is—rather, tool procedures should begin by selecting the tool upon which the user acted to ensure that it is working with the proper tool.

For example, a log procedure can begin as follows:

```
PROC MyLogProcedure(params)
    LogSelect(params[_ItemNum], params[_FormNum])
    .
    .
    .
END PROC
```

As previously explained, the `_ItemNum` and `_FormNum` elements of the passed tool procedure parameter array identify what tool invoked the procedure. These values are therefore used as parameters in the `LogSelect` command.

When several tools call the same tool procedure, the selection of the tool invoking a procedure ensures that all subsequent tool commands affect the correct tool. In addition, if a tool receives a number of messages (such as `_Click`, `_Close`, `_Size`, and so forth), it is also important that the procedure check to see exactly what message was received. This is usually done with a `SELECT` statement involving the `_Invoke` element of the tool procedure parameter array.

For example, the following procedure takes a different action depending on the value in the `_Invoke` element:

```
PROC MyFormProcedure(params)
  FormSelect(params[_FormNum])
  SELECT CASE params[_Invoke]
    CASE _Close
      'Process _Close message
    CASE _Click
      'Process _Click message
    CASE _Size
      'Process _Size message
  END SELECT
END PROC
```

The Application Procedure

CA-Realizer also allows you to create a special tool procedure to send messages to the main application window. This procedure is established in the same way as other tool procedures, but with the `AppSetProc` command rather than `TOOLSetProc`.

`AppSetProc` sets the procedure or macro to invoke when the main CA-Realizer window is resized or closed:

```
AppSetProc [(ProcName | ModuleName)]
```

ProcName is the name of a defined procedure to be run; *ModuleName* is a string containing the file name of the module to be run. (If neither is specified, any procedure already attached to the application is disassociated from it.)

Note that just like tool procedures, when an application procedure is called, CA-Realizer passes it an array (an *application procedure array*) that contains information about the action that invoked the procedure. The elements in an application procedure array are almost identical to those that are found in a tool procedure parameter array.

For example, the following application procedure (*AppProc*) uses the value stored in the `_Invoke` element in the *params* application procedure array in a `SELECT CASE` statement to choose between one of two actions:

```
'Force the window to remain maximized
'and prevent the user from closing it

PROC appProc(params)
    SELECT CASE params[_Invoke]
        CASE _Size
            SetSys(_Size, {_Maximize})
        CASE _Close
            params[_UseRealizer] = 0
    END SELECT
END PROC

AppSetProc(appProc)
```


If the value of the `_Invoke` element is `_Size`, the `SetSys` command is executed; if it is `_Close`, the procedure sets the `_UseRealizer` element to 0, forcing the window to remain open.

A Tool Example

Before moving on, here is an example of how to use the tool commands together. This example uses charts. Most of the other tools follow similar patterns.

```
' Manual\PG_09\TOOLPROC.RLZ

PROC ChartHandler(params)
  ChartSelect(params[_ItemNum], params[_FormNum])
  SELECT CASE params[_Invoke]
  CASE _Close
    INPUT "Closing the chart. Bye!";
  CASE _Click
    ChartText(params[_XPos], params[_YPos], \
      "Click", _Screen, _CTPos_Center + \
      _CTPos_Center)
  END SELECT
END PROC

MyChart = ChartQUnique
ChartNew(MyChart; "My First Chart")
ChartControl(_Size; _Center, _Center, 80 pct, \
  70 pct)
ChartLine(Sin(Index(360) * (_Pi/180)))
ChartSetColor(_Red)
ChartSetProc(ChartHandler)
ChartControl(_Show)
```

The tool procedure, called *ChartHandler*, is declared first (CA-Realizer requires that all procedures be declared before they are used). The chart procedure takes one parameter, *params*, which is the tool procedure parameter array.

ChartHandler first selects the chart on which the user acted, ensuring that the chart commands used afterwards affect that chart. The reason for this is that another chart may have been selected between the time the chart was created and the time that the user acted on it.

ChartHandler then uses a `SELECT` statement to determine the type of user action. If the user tries to close the chart, CA-Realizer displays a message before it is closed. If the user clicks in the chart, the word `click` is displayed in the chart at the point of the click.

The main program begins after the declaration of the chart procedure. First, a unique ID for the chart is generated with `ChartQUnique`; this number is then used with `ChartNew` to create a chart with the title "My First Chart." The chart is sized and centered with `ChartControl(_Size)`, and then a sine curve is plotted with the `ChartLine` command.

`ChartSetColor` changes the color of any further chart output, and then the chart's procedure is attached with the `ChartSetProc` command. Finally, the chart is made visible with the `ChartControl(_Show)` command.

When the program runs, a chart with a sine curve is displayed. At this point, CA-Realizer is not running a program. The user can move or resize the chart or can run another CA-Realizer program. If the user clicks in the chart or tries to close it, CA-Realizer runs the *ChartHandler* procedure, which warns the user as it closes the chart or displays text at the point of the click.

You can find this program—as well as the others in this part of the CA-Realizer Programming Guide—in the `\REALIZER\MANUAL` subdirectories.

Tool Command Summary

This section lists all available CA-Realizer *TOOL* commands and functions by tool.

Animator Commands and Functions

AnimateCells	AnimateControl
AnimateFrame	AnimateSelect
AnimateSpecialFrame	

Board Commands and Functions

BoardControl	BoardNew
BoardQ	BoardQUnique
BoardSelect	BoardSetProc
BoardUpdate	

Chart Commands and Functions

ChartBar	ChartControl
ChartControlGrid	ChartControlKey
ChartControlLabels	ChartControlPane
ChartHBar	ChartLine
ChartMark	ChartNew
ChartPie	ChartQ
ChartQPtInfo	ChartQUnique
ChartQXAxis	ChartQYAxis
ChartRemove	ChartSelect

ChartSelectPane	ChartSetColor
ChartSetFont	ChartSetKey
ChartSetLineStyle	ChartSetMark
ChartSetProc	ChartSetTitle
ChartSetXAxis	ChartSetXAxis2
ChartSetXGrid	ChartSetXLabels
ChartSetXView	ChartSetYAxis
ChartSetYGrid	ChartSetYLabels
ChartText	ChartUpdate
ChartXYLine	ChartXYMark

Form Commands and Functions

FormControl	FormModifyObject
FormNew	FormQ
FormQNum	FormQObject
FormQStr	FormQUnique
FormSelect	FormSetColor
FormSetGrid	FormSetObject
FormSetProc	FormWait

Graphics Tablet Commands and Functions

TabletBitmap	TabletBrush
TabletChord	TabletControl
TabletEllipse	TabletGetPixel
TabletLine	TabletLineTo
TabletMouseX	TabletMouseY
TabletMoveTo	TabletPen
TabletPie	TabletPolygon
TabletPolyline	TabletPolyPolygon
TabletRectangle	TabletRoundRect
TabletSelect	TabletSetColor
TabletSetPixel	TabletText
TabletTextColor	TabletUndo

Log Commands and Functions

LogControl	LogNew
LogQ	LogQData
LogQSize	LogQStr
LogQUnique	LogSelect
LogSetData	LogSetProc

Menu Commands and Functions

MenuControl	MenuDoCmd
MenuNew	MenuQ
MenuQUnique	MenuSelect
MenuSetCmd	MenuSetProc

Scheduler Commands and Functions

SchedControl	SchedNew
SchedQ	SchedQData
SchedRemove	SchedSet

Spreadsheet Commands and Functions

SheetControl	SheetControlLabels
SheetNew	SheetQ
SheetQInfo	SheetQUnique
SheetSelect	SheetSetColLabels
SheetSetProc	SheetSetRowLabels
SheetUpdate	

Chapter 10

Using Fonts and Pictures in Tools

In This Chapter	10-1
Fonts	10-2
Creating a Font	10-2
Generating an ID	10-3
Specifying an ID	10-3
Creating the Font	10-3
Selecting, Querying, and Closing a Font	10-4
Pictures	10-6
Creating a Picture	10-6
Generating an ID	10-6
Specifying an ID	10-7
Creating the Picture	10-7
Selecting, Querying, and Closing a Picture	10-9
Using the Clipboard	10-10
Copying a Picture from the Clipboard	10-10
Sending a Picture to the Clipboard	10-11
A Picture Example	10-11

Chapter 10

Using Fonts and Pictures in Tools

In This Chapter

CA-Realizer's programmable application tools can employ a variety of fonts to display text and numbers, and some tools—such as forms—can also display pictures. The use of fonts and pictures can greatly enhance the appearance of your applications.

Instead of specifying all of the required font or picture information each time you want to use one in your program, you can define a font or picture—and all of its associated characteristics—only once. The font or picture can then be used as many times as needed, by any of the appropriate tools.

The following table briefly describes the commands and functions that can be used to control font and picture information:

Command	Description
FontControl	Allows you to control the current font.
FontNew	Creates a new font definition.
FontQ	Returns information about a font.
FontQUnique	Generates a unique font ID to use with the FontNew command.
FontSelect	Selects the font to be affected by subsequent font commands.

Continued

Continued

Command	Description
PictureClipGet	Retrieves a bitmap or a metafile from the Clipboard and returns a new picture ID.
PictureClipSet	Copies a picture to the Clipboard.
PictureControl	Allows you to control the current picture.
PictureNew	Creates a new picture resource for use as an object in a form or a cell in an animation.
PictureQ	Returns information about a picture.
PictureQUnique	Returns a unique picture ID for use with the PictureNew command.
PictureSelect	Selects the picture to be affected by subsequent picture commands.

Many of these commands and functions are discussed in this chapter. For additional information, refer to the *CA-Realizer Language Reference Guide*.

Fonts

A *font* is a representation of a specific type face, point size, and style (such as bold, italics, underline, or strikeout). CA-Realizer allows you to access any fonts that are available on your system or printer.

Creating a Font

Use the FontNew command to create a font. With creating a font, you can either have CA-Realizer generate a unique font number (ID) for you, or you can specify a particular font ID.

You then use this font ID with any of the commands that accept a font as a parameter (for example, FormSetObject, ChartSetFont, and SheetControl).

Generating an ID

FontQUnique

You can use the `FontQUnique` function before issuing the `FontNew` command to generate a unique font ID. In this example, `FontQUnique` generates a new ID, passes it to the *FontID* variable, and then uses it in the next `FontNew` command:

```
FontID = FontQUnique
FontNew(FontID; "Helv", 12)
```

FontNew

You can also instruct the `FontNew` command to generate a unique ID. By simply omitting the optional *FontNum* parameter or by setting it to 0, a unique font number is generated automatically. (See [Creating the Font](#) below for details on the syntax of `FontNew`.)

Specifying an ID

To explicitly set a particular font ID, specify the desired number as the optional *FontNum* parameter in the `FontNew` command. (See [Creating the Font](#) below for details on the syntax of `FontNew`.)

Font IDs can be any number between 1 and 32767. If you specify an ID for a new font that is already used by an existing font, the old font is destroyed and the specified ID is set for the new font. For example, creating a Font 10 destroys any previous Font 10.

Creating the Font

The general syntax of the `FontNew` command is:

```
FontID = FontNew([FontNum]; TypeFace, PointSize
[, Style])
```

FontID is the variable to which the new font's ID is optionally passed. *FontNum* is the ID to be set for the new font. (If *FontNum* is 0 or omitted, a unique font number is generated automatically and returned to *FontID*.)

TypeFace is a string holding the name of the font (for example, "Courier", "Helv", or "Times New Roman"). *PointSize* is the desired font size, in points. (If the requested size is not available, CA-Realizer uses the closest available size that is not smaller than that requested.) *Style* can be any combination of the `_Bold`, `_Italics`, `_Underline`, or `_Strikeout` constants.

For example, the following command defines a 12-point bold and italic Script font as font 1:

```
FontNew(1; "Script", 12, _Bold + _Italics)
```

FontNew also accepts a string array (*FontDesc*) instead of a set of the *TypeFace*, *PointSize*, and *Style* parameters:

```
FontID = FontNew([FontNum]; FontDesc)
```

The first element in *FontDesc* is the font name, the second is the size, and the third is the style. This array is particularly handy when used in conjunction with the `StdFont` function, which returns an array in this form.

Selecting, Querying, and Closing a Font

Selecting a Font

By default, when you create a font with `FontNew`, it becomes the current font (this means that all subsequent font commands apply to it). To select a different current font, use `FontSelect`:

```
FontSelect (FontID)
```

FontID is the ID of the font that should now be the current font. For example, here is a typical command sequence for generating a series of related fonts:

```
Roman1 = FontNew(; "Times New Roman", 10)
Roman2 = FontNew(; "Times New Roman", 14, _Bold)
Roman3 = FontNew(; "Times New Roman", 14, _Italics)
.
.
.
'Select and use a particular font
FontSelect (Roman1)
.
.
'Destroy a font when you've finished with it
FontControl(_Close)
```

Querying a Font

You can query a font, like the other programmable application tools, to find out information about it (for example, what its defined typeface, size, or style is or whether it still exists). To do this, use the font-specific version of the *TOOLQ* command, *FontQ*:

```
Info = FontQ(Query [, FontNum])
```

The constant you supply for *Query* determines what type of information *FontQ* returns (available constants are listed in the *Language Reference Guide*).

By default, *FontQ* returns information about the currently selected font. However, you can specify another font's ID using the *FontNum* parameter to get information about that font.

For example:

```
'What is the face name of the current font?
Face = FontQ(_Title)

'Was font 10 defined?
IF Not(FontQ(_Exists; 10)) THEN
    PRINT "Undefined font"
END IF
```

This next example allows you to query the fonts that are currently installed on your system and then prints the results to the Print Log:

```
FormNew
FormSetObject(10, _ListBox, "", _Center, _Center; \\  
    _ListPrFonts)
FormControl(_Show)
PRINT FormQStr(10, 1)
```

Closing a Font

Use the *FontControl(_Close)* to remove the definition of the current font. If you need to use the font again, recreate it with the *FontNew* command.

Pictures

Bitmaps and metafile pictures can be used in CA-Realizer forms and animation windows. Pictures are extremely flexible, allowing pieces to be extracted and mirror images to be created.

Creating a Picture

Use the `PictureNew` command to create a picture. With creating a picture, you can either have CA-Realizer generate a unique picture number (ID) for you, or you can specify a particular picture ID.

You then use this picture ID with any of the commands that accept a picture as a parameter (for example, `FormSetObject` and `AnimateCells`).

Generating an ID

PictureQUnique

You can use the `PictureQUnique` function before issuing the `PictureNew` command to generate a unique picture ID. In this example, `PictureQUnique` generates a new ID, passes it to the *PictureID* variable, and then uses it in the next `PictureNew` command:

```
PictureID = PictureQUnique
PictureNew(PictureID; "arrow01.bmp")
```

PictureClipGet

Using the `PictureClipGet` command, you can copy an image from the Clipboard to create a new bitmap or metafile; a unique picture ID is returned for it. (If there is no image on the Clipboard, 0 is returned.)

Note: For more information about copying to or from the Clipboard, see *Using the Clipboard* later in this chapter.

PictureNew

You can also instruct the `PictureNew` command to generate a unique ID. By simply omitting the optional *PictureNum* parameter or by setting it to 0, a unique picture number is generated automatically. (See *Creating the Picture* below for details on the syntax of `PictureNew`.)

Specifying an ID

To explicitly set a particular picture ID, specify the desired number as the optional *PictureNum* parameter in the `PictureNew` command. (See *Creating the Picture* below for details on the syntax of `PictureNew`.)

Picture IDs can be any number between 1 and 32767. If you specify an ID for a new picture that is already used by an existing picture, the old picture is destroyed and the specified ID is set for the new picture. For example, creating a Picture 10 destroys any previous Picture 10.

Creating the Picture

Use the `PictureNew` command to create a new picture resource for use as an object in a form or a cell in an animation. Pictures can be loaded from a file or created as part of an existing picture. With `PictureNew`, you can:

- Load a bitmap file
- Load a metafile
- Copy a picture from the Clipboard
- Use all or part of an existing picture

The general form for the `PictureNew` command is shown below. Use the first `PictureNew` syntax to load a file from disk, and the second to create a new picture from all or part of an existing picture.

```
[PictureID] = PictureNew([PictureNum]; Name
                        [, Type])
[PictureID] = PictureNew([PictureNum]; Num
                        [, Type], Left, Top, Width, Height)
```

PictureID is the variable to which the new picture's ID is optionally passed. *PictureNum* is the ID to be set for the new picture. (If *PictureNum* is 0 or omitted, a unique picture number is generated automatically and returned to *PictureID*.)

Name is the file name of the desired bitmap or metafile, and *Type* specifies the type of picture to be created, and must be either `_Bitmap`—the default—or `_Metafile`.

For example:

```
'Create a new picture using a bitmap
pic = PictureNew(10; "C:\WINDOWS\LEAVES")
'Create a new picture using a metafile
mf = PictureNew(; "test.wmf", _Metafile)
```

In the second syntax (beginning with *PictureID*), *Num* is the picture ID of an existing picture. Use *Left*, *Top*, *Width*, and *Height* to specify what portion of the existing picture should be used to create the new picture. You can use the entire picture by specifying its full size (to get the width and height of the original picture, use the `PictureQ(_Size)` function), or a specific portion by specifying the desired area—*Left*, *Top*, *Width*, and *Height* are the coordinates of the portion to extract and must be specified in pixels.

For example:

```
'Create a new picture from the left-top quarter
'of an existing picture
newpic = PictureNew(0; pic, 0, 0, picSize[1] / 2, \
                    picSize[2] / 2)
```


If you specify negative values for *Width* and/or *Height*, you can create a mirror image of the picture. To create a vertical mirror image, specify a negative *Height* value; to create a horizontal mirror image, specify a negative *Width* value.

```
'Create a mirror image of an existing picture
mirror = PictureNew( a, 0, 0, -picSize[1], \
    picSize[2])
```

The ability to extract a portion of an existing picture is useful for manipulating several small bitmaps (for example, for use in an animation). The bitmaps can be saved together in one file and then extracted as separate pictures. This limits the number of actual bitmaps to one, reducing system resource utilization.

Selecting, Querying, and Closing a Picture

Selecting a Picture

By default, when you create a picture with `PictureNew`, it becomes the current picture. This means that all subsequent picture commands apply to it.

To set a different picture to be the current picture, use `PictureSelect`:

```
PictureSelect (PictureID)
```

in which *PictureID* is the ID of the picture that should now be the current picture.

Querying a Picture

You can query a picture, like the other programmable application tools, to find out information about it (for example, what its size is or whether it is a bitmap or metafile). To do this, use the picture-specific version of the `TOOLQ` command, `PictureQ`:

```
Info = PictureQ(Query [, PictureNum])
```

The constant you supply for *Query* determines what type of information `PictureQ` returns (available constants are listed in the *Language Reference Guide*).

By default, `PictureQ` returns information about the currently selected picture. However, you can specify another picture's ID using `PictureNum` to get information about that picture.

For example:

```
'What is the size of the current picture?
Size = PictureQ(_Size)
```

Closing a Picture

Use the `PictureControl(_Close)` to remove the definition of the current picture. If you need to use the picture again, recreate it with the `PictureNew` command.

Using the Clipboard

CA-Realizer provides two commands that allow you to work with your system's Clipboard. `PictureClipGet` allows you to place a copy of the data currently stored in the Clipboard in a CA-Realizer form or animation window; conversely, `PictureClipSet` allows you to copy a picture to the Clipboard.

Copying a Picture from the Clipboard

If there is already a bitmap or metafile picture in the Clipboard, you can retrieve it from the Clipboard using the `PictureClipGet` command as follows:

```
[PictureNum] = PictureClipGet([Format])
```

Format can be either `_Bitmap` (the default) or `_Metafile`. This command assigns a unique picture ID to the new picture.

For example, the following retrieves a bitmap from the Clipboard and pastes it in a form:

```
pic = PictureClipGet
IF pic THEN
    FormSetObject(10, _Bitmap, pic, 10, 10)
END IF
```

Sending a Picture to the Clipboard

A picture can be copied to the Clipboard using the `PictureClipSet` command. The picture is automatically placed in the Clipboard in the format it was originally created (`_Bitmap` or `_Metafile`).

```
[Result] = PictureClipSet([PicNum])
```

By default, `PictureClipSet` copies the currently selected picture to the Clipboard. However, you can specify another picture's ID using the `PicNum` parameter to copy that picture.

If the picture is successfully placed, `PictureClipSet` returns a 1; otherwise, it returns 0.

For example, the following creates a picture and attempts to copy it to the Clipboard:

```
PictureNew(10; "C:\WINDOWS\CARS")
IF Not(PictureClipSet) THEN
    INPUT "Can't put picture in Clipboard";
END IF
```

If the picture cannot be copied, an error message is displayed.

A Picture Example

This example demonstrates how you might use of several of the picture commands in a form:

```
' Manual\PG_11\PIGBANK.RLZ

'Create a form to hold the bitmap
FormNew(1)
FormSetColor(_LightGray; _Field)

'Create a picture
PicNum = PictureQUnique
PictureNew(PicNum; "CLIPART\BANK05.BMP", \
    _Bitmap)

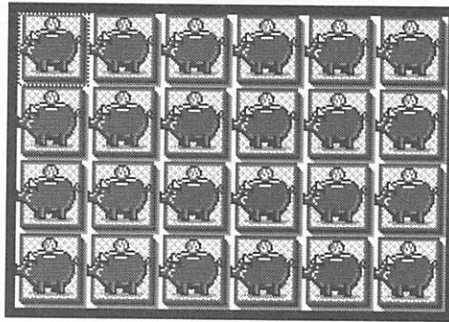
'Query the size of the first picture
PicSize = PictureQ(_Size)
PicWidth = PicSize[1]
PicHeight = PicSize[2]

'Resize the form to hold a 6x4 array of pictures
FormControl(_SizeInside; _Center, _Center, \
```

```
        6 * PicWidth, 4 * PicHeight)
'Put the pictures in the form
object = 1
FOR row = 0 to 3
    FOR col = 0 to 5
        FormSetObject(object, _Picture, PicNum, \\  
                        col * PicWidth, row * PicHeight)
        object = object + 1
    NEXT col
NEXT row

'Show the form
FormControl(_Show)
```

The following figure displays the output generated by this code:



Chapter 11

Charts

In This Chapter	11-1
The Anatomy of a Chart	11-4
Creating a New Chart	11-5
Creating the Chart	11-5
Setting the Chart Type and Displaying the Chart	11-6
Available Plot Types	11-7
Bar Charts	11-8
Horizontal Bar Charts	11-11
Line and Marker Charts	11-12
XY Charts	11-16
Pie Charts	11-17
Controlling the Appearance of a Chart	11-19
Setting Chart Colors	11-19
Choosing Colors	11-22
Creating a Chart with Varied Colors	11-23
Setting Line Styles	11-24
Setting Chart Axes	11-26
The X-Axis	11-27
Setting the X-Axis with Date-Time Values	11-27
Setting a Secondary X-Axis	11-30
Setting the View for the X-Axis	11-31
Setting the Y-Axis	11-32
Adding a Chart Key	11-33
Adding Items to a Chart Key	11-33
Showing the Chart Key	11-33
A Sample Chart Key	11-35

Labeling Charts	11-37
Setting Chart and Axes Titles	11-37
Changing Axes Labels	11-38
Hiding Labels	11-39
Placing Text Within a Chart	11-40
Controlling the Chart Grid	11-43
Hiding and Showing the Grid Lines	11-43
Creating a Custom Grid	11-43
A Sample Custom Grid	11-44
Setting Fonts for a Chart	11-45
Advanced Charting Features	11-48
Updating a Plot	11-48
Removing a Plot	11-49
Querying Axis Values	11-49
Creating Multiple Panes	11-50
Controlling the Redraw	11-52
Setting a Chart Procedure	11-53
Attaching a Chart Procedure	11-53
Processing in the Chart Procedure	11-54
Converting Screen-Relative to Data-Relative Positions	11-54
A Sample Chart Procedure	11-55
Printing a Chart	11-57

Chapter 11

Charts

In This Chapter

Charts are one of the most versatile and effective tools for presenting information. CA-Realizer provides commands for creating many different styles of charts, as well as for controlling their content and format. In this chapter, you will learn how to use the chart tool for the visual display of data.

Note: The commands you use to create and manipulate a chart—such as `ChartNew`, `ChartQUnique`, `ChartControl`, and `ChartSelect`—work like their other tool counterparts. See the “Overview of the CA-Realizer Programmable Application Tools” chapter for details about the standard tool commands.

The following table briefly describes the commands and functions that can be used to create and control charts:

Command	Description
<code>ChartBar</code>	Adds a bar plot to the current chart pane.
<code>ChartControl</code>	Allows you to control the attributes of the current chart.
<code>ChartControlGrid</code>	Shows and hides grid lines in the current chart pane.
<code>ChartControlKey</code>	Adjusts the attributes of the key in the current chart.
<code>ChartControlLabels</code>	Controls the display of titles and axes.

Continued

Continued

Command	Description
ChartControlPane	Allows you to control the attributes of the current chart pane.
ChartHBar	Adds a horizontal bar plot to the current chart pane.
ChartLine	Adds a line plot to the current chart pane.
ChartMark	Adds a marker plot to the current chart pane.
ChartNew	Creates a new chart.
ChartPie	Adds a pie chart plot to the current chart pane.
ChartQ	Returns information about a chart.
ChartQPtInfo	Converts screen-relative coordinates to data-relative coordinates.
ChartQUnique	Returns a unique chart number.
ChartQXAxis	Queries a chart and returns the actual range of values being used for the x-axis.
ChartQYAxis	Queries a chart and returns the actual range of values being used for the y-axis.
ChartRemove	Removes a plot from the current chart.
ChartSelect	Selects a chart to be the current chart.
ChartSelectPane	Selects a pane to be the current chart pane.
ChartSetColor	Sets the color for elements of the current chart.
ChartSetFont	Sets the fonts for elements of the current chart.
ChartSetKey	Adds an item to the key of the current chart.

Continued

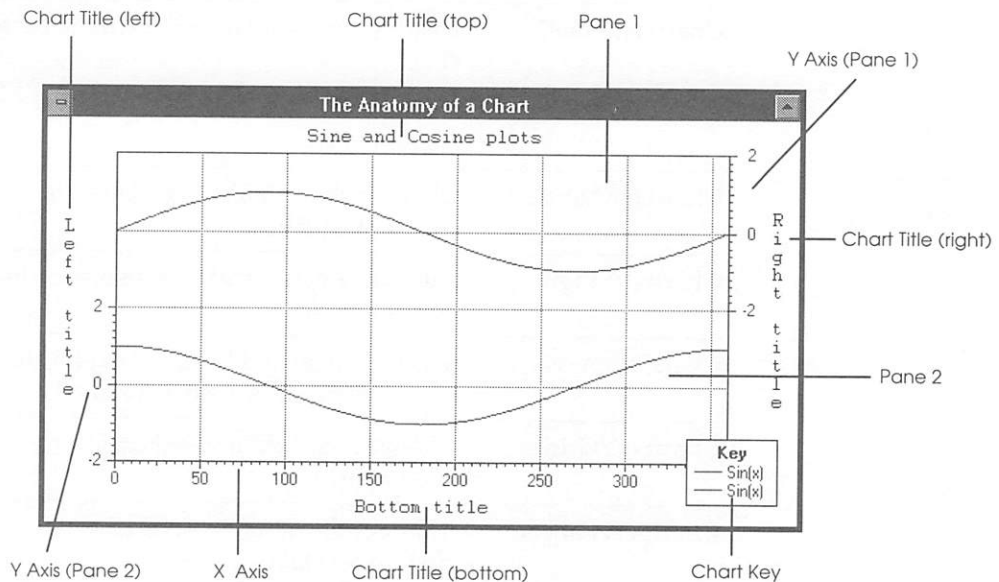
Continued

Command	Description
ChartSetLineStyle	Sets the pattern and width of lines plotted in the current chart
ChartSetMark	Sets the marker used in a marker plot.
ChartSetProc	Assigns a procedure to the current chart.
ChartSetTitle	Sets the text for titles in the current chart.
ChartSetXAxis	Sets the range of values shown on the x-axis of the current chart.
ChartSetXAxis2	Displays a secondary x-axis in the current chart.
ChartSetXGrid	Changes the default x-axis grid to a custom grid.
ChartSetXLabels	Changes the default x-axis labels to custom text labels.
ChartSetXView	Sets the range of x values displayed in the current chart.
ChartSetYAxis	Sets the range and label format for the y-axis of the current chart pane.
ChartSetYGrid	Changes the default y-axis grid to a custom grid.
ChartSetYLabels	Changes the default y-axis labels to custom text labels.
ChartText	Plots text labels at specific locations in the current chart pane.
ChartUpdate	Updates the contents of a plot.
ChartXYLine	Adds a line plot connecting a series of (x, y) pairs.
ChartXYMark	Adds a marker plot determined by a series of (x, y) pairs.

Many of these commands and functions are discussed in this chapter. For more detailed information, refer to the *CA-Realizer Language Reference Guide*.

The Anatomy of a Chart

A chart is comprised of many parts, most of which can be controlled separately, as discussed in this chapter. The diagram below outlines some basic terms associated with the chart:



Creating a New Chart

As discussed in the “Overview of the CA-Realizer Programmable Application Tools” chapter, a tool can be created using a *TOOLNew* command—in the case of charts, use the *ChartNew* command.

Creating the Chart

Like any CA-Realizer tool, each chart must have a unique ID. You can use the *ChartNew* or *ChartQUnique* command to generate one, or you can specify a particular ID when issuing the *ChartNew* command. For example:

```
'Use ChartQUnique to select the Chart number
'and then make the chart visible
ChartNew(ChartQUnique; "Another Chart", _Title)
ChartControl(_Show)
```

The general form of the *ChartNew* command is as follows:

```
[ChartID = ] ChartNew([ChartNum] [, Title \\  
[, Style]])
```

Use *Title* and *Style* to override the chart's default attributes. The chart's title is initially set to “Chart *ChartNum*” (for example, Chart 10). Specifying *Title* allows you to set it to a different text string.

In addition, the default window style of a chart is *_Title + _Size + _Minimize + _Close*. Using the *Style* parameter, you can add additional styles or remove some or all default ones. (Available *Style* constants are listed in the *Language Reference Guide*.)

For example:

```
'Create a chart with a frame so it cannot be moved
ChartNew(1; "Fixed Chart", _Frame)
```

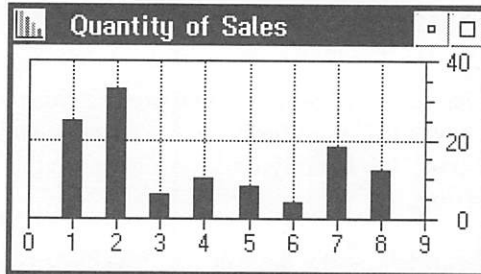
Setting the Chart Type and Displaying the Chart

After a chart is created, you must use the `ChartBar` command to indicate the type of plot to make and provide the appropriate specifications and data, and then the `ChartControl` command to size and display the chart.

For example, the following creates a bar chart that displays the values stored in the *Quantity* array:

```
Quantity = {25, 33, 6, 10, 8, 4, 18, 12}
ChartNew( "Quantity of Sales")
ChartBar(Quantity; 10)
ChartControl(_Size; _Center, _Center, \\  
            50 pct, 50 pct)
ChartControl(_Show)
```

These commands display a new tool window with the title "Quantity of Sales." The bar chart displayed in the window represents the value of each element in the *Quantity* array with a single bar. Index numbers for each element are listed along the x-axis, and the y-axis is scaled to fit the data. For example:



Note: The `ChartBar` and `ChartControl` commands are presented in greater detail later in this chapter.

Available Plot Types

The bar chart created in the previous example is only one type of plot. CA-Realizer also has six other plot types—each chart type is created by its own function, as listed in the table below:

Command	Description
ChartBar	Draws a vertical bar for each data point, and can be used to create high-low-close, I-Beam, and Candlestick charts.
ChartHBar	Draws a horizontal bar for each data point.
ChartMark and ChartLine	Draws a marker and/or line at each data point.
ChartPie	Draws a pie chart.
ChartXYLine	Draws a line connecting a list of (x, y) pairs.
ChartXYMark	Draws a marker at each (x, y) pair.

Each of these functions returns a *plot number*, which you can use in subsequent commands to update and remove those plots. You can also ignore the return values by simply calling the function as a procedure and not assigning the return value to a variable. The examples that follow illustrate both these methods.

Bar Charts

Use the `ChartBar` function to create a *bar* chart and two variants—the *Candlestick* chart and the *I-Beam* chart. `ChartBar` returns the plot number of a chart and accepts a number of different parameters:

```
[PlotNumber = ]ChartBar(HighArray [, LowArray  
                        [, CloseArray [, OpenArray]]]  
                        [, _Mask, BooleanArray]  
                        [, Dates][; Type][, Width1 [, Width2]])
```

Use *Type* to specify the type of bar chart—`_VertBar` (the default), `_Candlestick`, or `_IBeam`.

Width1 is the width of the vertical line (the default is 1 pixel). *Width2* is the total width of the bar, including ticks (the default is 7 pixels, 1 for the bar and 3 each for ticks).

The Bar Chart

You can create a basic bar chart by specifying a single array containing numeric values. The bar chart created in this example has a bar width of 10:

```
ChartBar(Quantity; 10)
```

Note: The width of the bars is always measured in pixels.

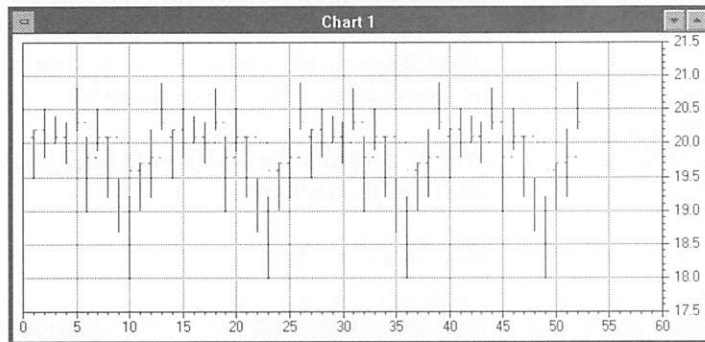
Notice that in this example, you call `ChartBar` in the way you call a procedure. CA-Realizer interprets this as meaning that you want to discard the returned *PlotNumber*.

You can just as easily create a high-low chart (commonly known as a “bar chart” in the financial community). For instance, you can plot stock data that contains a high price, low price, opening price, and closing price over a period of time.

The `STOCK.DAT` file in the `MANUAL` directory contains some sample data to plot:

```
FileImport("MANUAL\STOCK.DAT", _Realizer, _Named)  
ChartNew  
ChNum = ChartBar(HighPrice, LowPrice, ClosePrice, \  
                OpenPrice)  
ChartControl(_Show)
```

This code produces the following high-low-open-close bar chart:



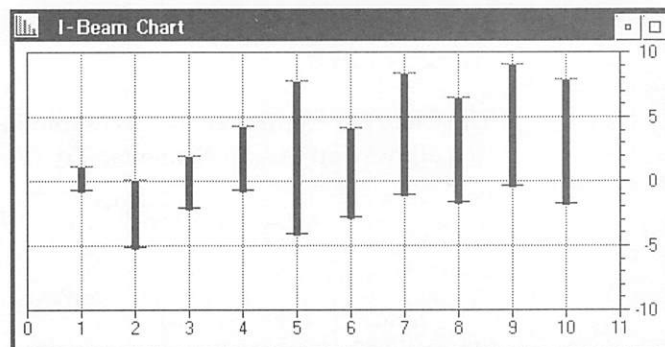
The I-Beam Chart

I-Beam charts are similar to high-low-open-close charts, except that the open and close ticks extend equally from left to right. *Width1* controls the width of the bar, and *Width2* controls the extension of the beam.

The following demonstrates how to create an I-Beam chart:

```
ChartNew("I-Beam Chart")
hi = Index(10) + (Rnd(10) - 0.5) * 6.0
close = hi
lo = hi - Rnd(10) * 11.0 - 1
open = lo
ChartBar(hi, lo, close, open; _IBeam, 5, 15)
ChartControl(_Show)
```

It creates two arrays—*hi* and *lo*—using *Rnd* and *Index* and assigns them to the *open* and *close* variables. Then, *ChartBar* creates an I-Beam plot with 5 pixel-wide bars and 15 pixel-wide extensions on each bar, producing the following I-Beam chart:

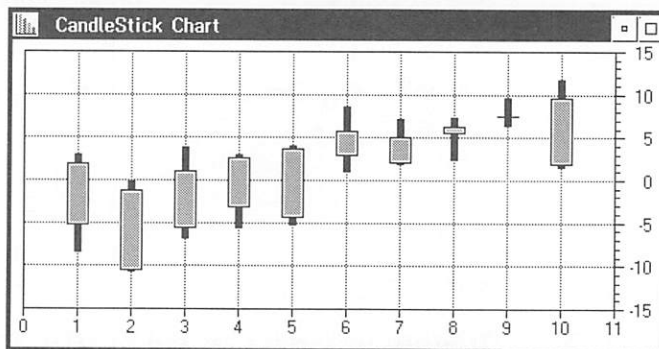


The Candlestick Chart Candlestick charts are similar to I-Beam charts, except that the beams are connected to form a rectangle. The convention for creating a Candlestick chart is that the candlestick rectangle is filled if *close* > *open*; otherwise, it is empty. (This is implemented as two colors, *_Data2* and *_Data3*.)

Width1 controls the width of the bar, and *Width2* controls the width of the rectangle.

Here is an example:

```
CandleStick = ChartQUnique
ChartNew(CandleStick; "CandleStick Chart")
Hi = Index(10) + (Rnd(10) - 0.5) * 6.0
Lo = Hi - Rnd(10) * 11 - 1
ChartBar(Hi, Lo, Hi - Rnd(10)*3, Lo + Rnd(10)*3; \
_Candlestick, 5, 15)
ChartControl(_Show)
```



Try changing the values of *open* and *close* and compare the differences:

```
ChartBar(Hi, Lo, Hi + Rnd(10)*3, Lo - Rnd(10)*3; \
_Candlestick, 5, 15)
ChartControl(_Show)
```

The *ChartBar* command also accommodates a Boolean vector that allows you to skip elements. Try the following:

```
BoolArray = {1, 1, 1, 0, 0, 1, 0, 1, 0, 1}
Hi = Index(10) + (Rnd(10) - 0.5) * 6.0
Lo = Hi - Rnd(10) * 11.0 - 1
ChartBar(Hi, Lo, Hi - Rnd(10)*3, \
Lo + Rnd(10)*3, _Mask, BoolArray; \
_Candlestick, 5, 15)
ChartControl(_Show)
```


Horizontal Bar Charts

Horizontal bar charts are particularly useful for creating Gantt charts that you use in project management and timeline schedules:

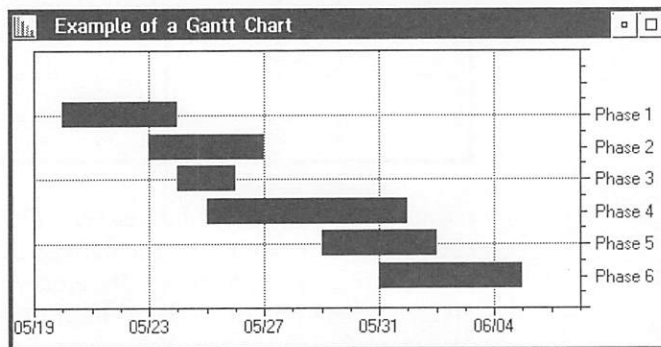
```
PlotNumber = ChartHBar(HighArray [, LowArray
    [, CloseArray [, OpenArray]]]
    [, _Mask, BooleanArray]
    [, [Type,] [Width1 [, Width2]]])
```

Horizontal bars are drawn on the same axes as the other charts. The major differences are that the array elements in the data represent the y values, not the x values. *Type* can also be `_VertBar`, `_IBeam`, or `_Candlestick`.

Here is an example of a Gantt chart:

```
Dv = QDate + Index(20)
Lo = {13, 11, 7, 6, 5, 2}
Hi = {18, 15, 14, 8, 9, 6}
Labels = {"Phase 6", "Phase 5", "Phase 4", "\
    "Phase 3", "Phase 2", "Phase 1"}

ChartNew(0; "Example of a Gantt Chart")
ChartSetXAxis(Dv, "D(M1/D1)", 1)
FOR i = 1 to 6
    ChartHBar(Hi[i:i], Lo[i:i]; 15)
NEXT i
ChartSetYLabels(Labels)
ChartControl(_Show)
```



Note the use of the `ChartSetXAxis` and `ChartSetYLabels` commands to control the axes—you will be taking a closer look at these commands later on.

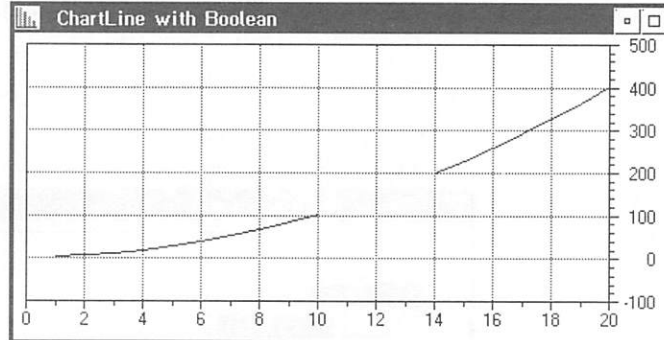
Line and Marker Charts

ChartMark and ChartLine plot a set of discrete data points, except that ChartLine connects the points with lines.

Both ChartLine and ChartMark take a Boolean array. If you do not specify the Boolean array, CA-Realizer plots all data in *Data* and scales the x-axis according to the valid range of the array. CA-Realizer scales the y-axis according to the largest and smallest data elements.

If you supply a Boolean array, lines will only be drawn from points for which the corresponding element in the Boolean array is non-zero. This allows for breaks in the chart as demonstrated below:

```
ChartNew(0;"ChartLine with Boolean")
yVals = Sqr(Index(20))
boolVals[1:9] = 1
boolVals[14:20] = 1
ChartLine(yVals, boolVals)
ChartControl(_Show)
```



If you supply an array of dates with *Dates*, CA-Realizer correlates each element in the data array to the corresponding time value in the date array. It determines the x position for a given element *i* by matching *Dates[i]* to the x-axis date array as set by the ChartSetXAxis command.

ChartLine

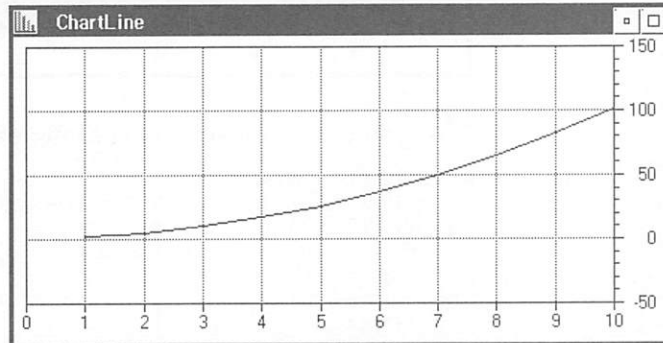
ChartLine plots all the points in an array, connecting each pair of adjacent points with a line. The full form for the ChartLine function is:

```
[PlotNum = ] ChartLine (Data [, Bool]
    [, Dates] [, _Fill [, FillTo [, Value]]])
```

The index of the array is the x-coordinate, and the value corresponding to that index is the y-coordinate.

For example, the following command statements plot the squares of the numbers 1 to 10:

```
ChartNew(ChartQUnique; "ChartLine")
ChartLine(Sqr(Index(10)))
ChartControl(_Show)
```

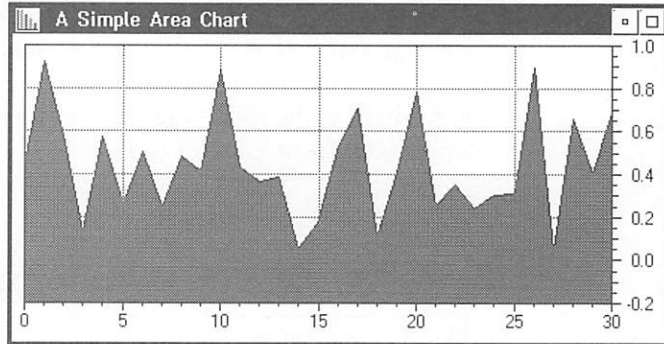


The ChartLine function also takes a modifier, which allows the region beneath the line (or between the line and another line) to be filled. You can easily create filled line charts by specifying the `_Fill` modifier with one of the following *FillTo* values:

Constant	Description
<code>_Down</code>	Fills to the bottom of the pane. This is the default.
<code>_Up</code>	Fills to the top of the pane.
<code>_ToPlot</code>	Fills to the specified plot determined by <i>Value</i> (a plot number).
<code>_ToValue</code>	Fills to the y value specified by <i>Value</i> (the default is 0).

An example of a filled area chart follows:

```
AreaChart = ChartQUnique  
ChartNew(AreaChart; "A Simple Area Chart")  
Values[0:30] = Rnd(31)  
ChartLine(Values; _Fill, _Down)  
ChartControl(_Show)
```



Now you can try the following changes:

```
AreaChart = ChartQUnique  
ChartNew(AreaChart; "A Simple Area Chart")  
Values1[0:30] = Rnd(31)  
Values2[0:30] = Rnd(31)+ 1  
plotNum = ChartLine(Values1)  
ChartLine(Values2; _Fill, _ToPlot, plotNum)  
ChartControl(_Show)
```

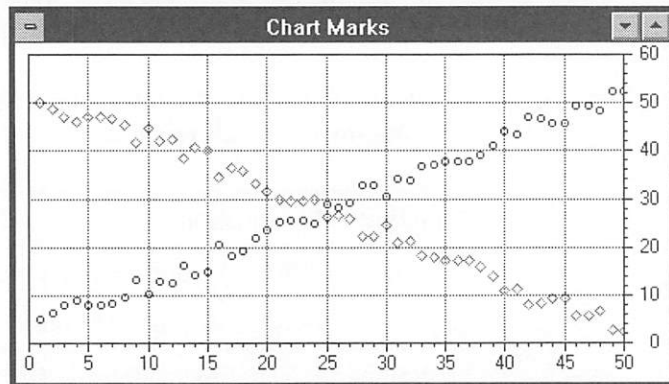
ChartMark

ChartMark is similar to ChartLine, except that it plots a marker at each data point instead of connecting the points with lines. The default marker is a circle:

```
[PlotNum = ] ChartMark(Data [, Bool]
    [, Dates])
```

You can select different markers with the ChartSetMark command:

```
ChartNew(ChartQUnique)
DataUp = Index(50) + Rnd(50) * 5
DataDown = 55 - DataUp
ChartMark(DataUp)
ChartSetColor(_Red)
ChartSetMark(_Diamond)
ChartMark(DataDown)
ChartControl(_Show)
```



Note: The *Bool* and *Dates* parameters work the same as they do with ChartLine.

There are a wide variety of marker types that can be set with the ChartSetMark command. For more information, refer to the *CA-Realizer Language Reference Guide*.

XY Charts

`ChartXYMark` and `ChartXYLine` plot points whose position is determined by two arrays: one for the x-coordinates and one for the y-coordinates.

```
PlotNumber = ChartXYMark(XCoords, YCoords  
    [, BooleanArray])
```

```
PlotNumber = ChartXYLine(XCoords, YCoords,  
    [, BooleanArray][, DateArray]  
    [, _Fill [, FillTo [, Value]]])
```

`ChartXYLine` is useful for drawing figures. It uses two arrays—`XCoords` contains the x-coordinates and `YCoords` contains the y-coordinates. A line connects each (x, y) pair. (x[1], y[1]) is connected to (x[2], y[2]), which is connected to (x[3], y[3]), and so on. If you supply `BooleanArray`, lines will only be drawn from points for which `BooleanArray` is non-zero.

Filled XY Line charts also take a modifier specified by `FillTo`. It can take any of the following values:

Constant	Description
<code>_Down</code>	Fills to the bottom of the pane.
<code>_Up</code>	Fills to the top of the pane.
<code>_ToValue</code>	Fills to the y value specified by <i>Value</i> (the default is 0).
<code>_Inside</code>	Closes the polygon created by the data by connecting the last point back to the first, and then fills in the bounding area with the <code>_Data2</code> color.

Try the following code fragment:

```
ChartNew(, "An XY Area Chart")  
ChartSetColor(_Red)  
Values1[0:30] = Index(31)  
Values2[0:30] = Rnd(31) + 1  
ChartXYLine(Values1, \  
    Values2; _Fill, _ToValue, 1.5)  
ChartControl(_Show)
```

ChartXYMark is similar to ChartXYLine, except that it draws a marker at each (x, y) pair instead of drawing a connecting line.

Pie Charts

You can use the ChartPie command to create pie charts:

```
PlotNumber = ChartPie(DataArray,
    [, LabelsArray][, ColorsArray]
    [, CenterX, CenterY, Radius]
    [, Style])
```

DataArray is the array of values to plot, *LabelsArray* is the array of corresponding labels to attach to the chart (the default is none), and *ColorsArray* is the array of colors to use for each element of the chart (the defaults are sequential).

If specified, *CenterX* and *CenterY* set the center of the pie in data (x-axis) coordinates, while *Radius* sets the radius of the pie. If you do not specify *CenterX*, *CenterY*, and *Radius*, CA-Realizer places the pie in the center and does not add the axes.

Style can be one or a sum of the following:

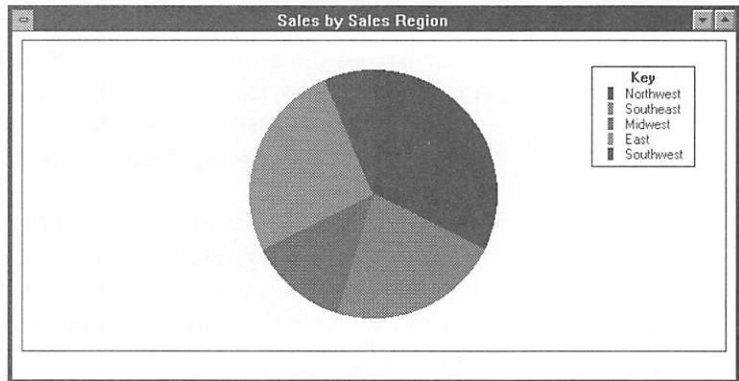
Constant	Description
_Labels	Puts the labels on the pie chart.
_NoLabels	Does not put the labels on the pie chart itself.
_KeyLabels	Automatically adds the labels to the chart key.

Note: By default, labels are only added to the pie chart. The y-axis will not be affected by the pie chart, and the pie may be clipped within the chart.

Here is a typical pie chart:

```
Values = {10, 7, 4, 8, 2}
Labels = {"Northwest", "Southeast", \\  
         "Midwest", "East", "Southwest"}
ChartNew(0;"Sales by Sales Region")
ChartPie(Values, Labels; _NoLabels + _KeyLabels)
ChartControlKey(_Size; 80 pct, 10 pct)
ChartControlKey(_Show)
ChartControl(_Show)
```

It displays the following pie chart:



Notice the use of the ChartPie modifiers, `_NoLabels` and `_KeyLabels`. These tell CA-Realizer to suppress placing labels around the perimeter of the pie chart and to place them in a key instead. The use of the ChartControlKey command will be reviewed later in this chapter.

Controlling the Appearance of a Chart

The `ChartControl` command works the same as all the other `TOOLControl` commands, allowing you to show, hide, minimize, maximize, size, and print the chart. (Refer to “Overview of the CA-Realizer Programmable Application Tools” in this guide for more information.)

CA-Realizer also provides a wide range of commands and functions for changing the appearance of the data in the chart. These include commands for setting fonts, selecting line colors, line styles and line markers, creating keys, and manipulating the axes, grids, and titles.

Note: As with the other CA-Realizer tools, when you create a chart, it becomes the current chart and all subsequent chart commands affect it. Use the `ChartSelect` command to select a different chart as the current chart.

Setting Chart Colors

Color is useful for charts that display more than one set of data. You can use color to differentiate between different data plots and to create a chart key that indicates how data values and colors are associated.

Before plotting data, you can specify how to use colors in a chart with the `ChartSetColor` command. `ChartSetColor` sets the color you use to draw the different parts of a chart. You can specify different colors for almost every aspect of a chart.

The general form of `ChartSetColor` is as follows:

```
ChartSetColor(Color [; Which])  
ChartSetColor(Red, Green, Blue [; Which])
```

Color and *Red*, *Green*, *Blue* represent the new color to be set. (See [Choosing a Color](#) below for more information.)

Which describes the element for which a color should be set, and can be any one of the following:

Constant	Description
_ChartBackgr	Chart background
_PaneBackgr	Pane background
_KeyBackgr	Key background
_Background	Chart background, all pane backgrounds, and key background
_Data1	Data in the chart (lines, markers, and vertical lines in bars)
_Data2	Data in the chart (close ticks for bars and filled area in line charts)
_Data3	Data in the chart (open ticks for bars, and filled area in line charts)
_AllData	All data in the chart
_LeftTitle	Left title
_TopTitle	Top title
_RightTitle	Right title
_BottomTitle	Bottom title
_KeyTitle	Key title
_AllTitle	All the chart titles and key title
_XAxis_Color	X-axis labels
_XAxis2	Secondary x-axis labels
_YAxis_Color	Y-axis labels for the current pane
_AllYAxis	Y-axis labels for all panes

By default, the various chart elements are drawn in the following colors:

Color	Element
_White	_ChartBackgr, _KeyBackgr, _PaneBackgr
_Black	_KeyBorder, _Grid, _KeyText, _LeftTitle, _TopTitle, _RightTitle, _BottomTitle, _KeyTitle, _XAxis_Color, _XAxis2, _YAxis_Color
_Blue	_Data1, _ChartText
_Green	_Data2
_Red	_Data3

Note: When the color of a fixed chart element (such as a chart title or the x-axis) is changed, the change is reflected immediately. When a data color is changed, it only affects subsequent plot commands that are based on that color.

Choosing Colors

When deciding what color to use with `ChartSetColor`, you can choose a predefined color or you can create your own.

Predefined Colors

The predefined colors of CA-Realizer can be seen below:

_Black	_Evergreen	_PowderBlue	_MediumBrown
_White	_MediumBlue	_LightYellow	_DarkGreen
_Gray	_Purple	_LimeGreen	_DarkGreenBlue
_Red	_Turquoise	_LightGreen	_NavyBlue
_Green	_Brown	_Maroon	_DarkPurple
_Blue	_Pale	_Pink	_Tan
_Cyan	_Cream	_Orange	_GrayGreen
_Magenta	_GreenYellow	_LightEvergreen	_LightGray
_Yellow	_MediumGreen	_DarkBrown	_Violet
_Brick	_BlueGreen		

For example:

```
ChartSetColor(_Red; _Data1)
```

Custom Colors

If the colors provided by CA-Realizer are not sufficient for your needs, you can create a new color by describing it as an RGB (Red-Green-Blue) value. In this case, you should set the *Red*, *Green*, and *Blue* parameters in the `ChartSetColor` command to the desired red, green, and blue intensities (where 0 represents no color and 1 represents full intensity). For example, 0.0, 0.0, 0.0 is black; 0.5, 0.0, 0.0 is a medium intensity red; and 0.8, 0.8, 0.8 is a light gray.

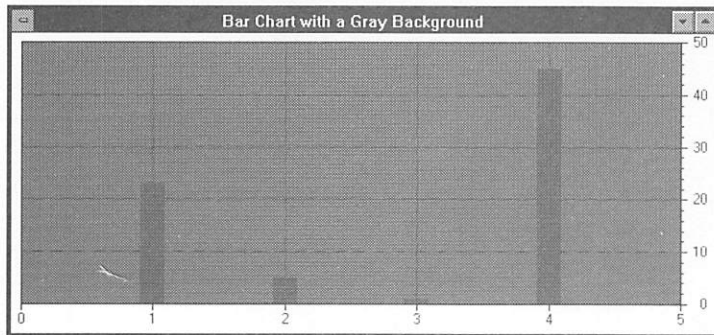
Tip: To quickly obtain the RGB values that make up one of the predefined colors, use the `QSys(_Color)` function.

Note: If a monitor cannot show the requested color, CA-Realizer displays the closest possible color.

Creating a Chart with Varied Colors

This example creates a bar chart with red bars, blue labels, and a gray background:

```
NumSold = {23, 5, 1, 45}
ColorChart = ChartQUnique
ChartNew(0; "Bar Chart with a Gray Background")
ChartSetColor(_Red; _Data1)
ChartSetColor(_Blue; _AllText)
ChartSetColor(_Gray; _PaneBackgr)
ChartBar(NumSold; 20)
ChartControl(_Size; _Center, _Center, 80 pct, \
50 pct)
ChartControl(_Show)
```

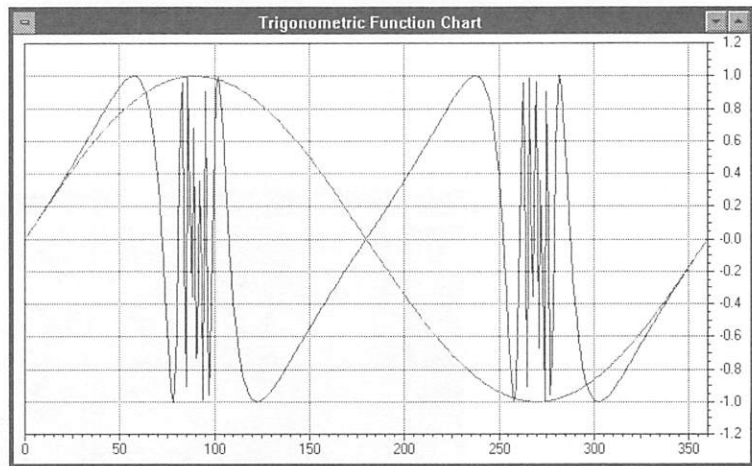


The three `ChartSetColor` commands control the color of all the text in the chart, the background color of the chart pane, and the color of the bars you use to display the data. You can also sum these modifiers. For example, the following command sets the color of all text, as well as all data, to red:

```
ChartSetColor(_Red; _AllData + _AllText)
```

Try the following multi-color line chart that is generated using some of the trigonometric functions of CA-Realizer:

```
' Manual\PG_11\TRIG1.RLZ  
Radians = (3.141593/180) * Index(360)  
SinTanWave = Sin(Tan(Radians))  
SinWave = Sin(Radians)  
Waves = ChartNew(, "Trigonometric Function Chart")  
ChartSetColor(_Blue)  
ChartLine(SinTanWave)  
ChartSetColor(_Red)  
ChartLine(SinWave)  
ChartControl(_Show)
```



Setting Line Styles

By default, lines in a chart are solid and one pixel wide. You can vary the pattern and width of chart lines using the `ChartSetLineStyle` command:

```
ChartSetLineStyle(Style [, Width])
```

where *Style* determines the pattern of the line, and *Width* specifies the width of the line, in pixels.

Note: If a line is assigned a pattern, it can only be one pixel wide.

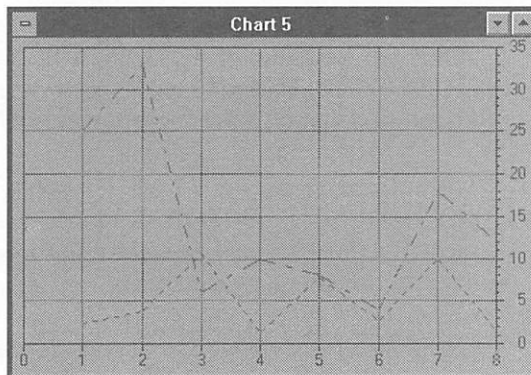
Style can be any one of the following:

Constant	Line Patterns
<code>_Solid</code>	—————
<code>_Dash</code>	—— ———
<code>_Dot</code>	-----
<code>_DashDot</code>	- - - - -
<code>_DashDotDot</code>	- . - . - .

You can build a new line chart with two data sets, differentiated by line style. Set the line style before adding each line to the chart:

```
Quantity = {25, 33, 6, 10, 8, 4, 18, 12}
Price = {2.3, 3.75, 10.5, 1.25, 7.75, \
        2.65, 9.95, 1.45}

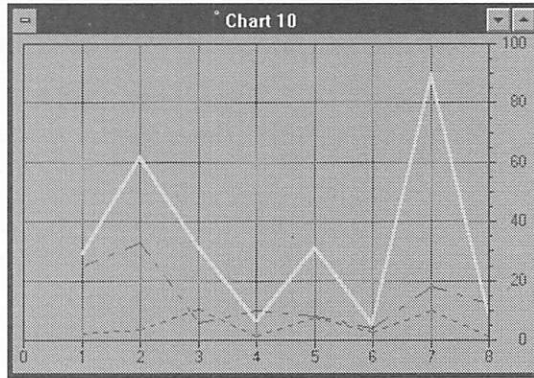
ChartNew
ChartSetColor(_Red; _Data1)
ChartSetColor(_Blue; _AllText)
ChartSetColor(_LightGray; _Background)
ChartSetLineStyle(_DashDot, 1)
ChartLine(Quantity)
ChartSetColor(_White; _Data2)
ChartSetLineStyle(_Dot, 1)
ChartLine(Price)
ChartControl(_Size; _Center, _Center, 50 pct, \
            50 pct)
ChartControl(_Show)
```



CA-Realizer defines the thickness or width of a line with the second parameter of `ChartSetLineStyle` and measures it in pixels.

Add these lines to the line chart you just created (*after* the `ChartLine(Price)` command and *before* the first `ChartControl` command) to add a new line that is different from the others:

```
ChartSetColor(_Yellow)
ChartSetLineStyle(_Solid, 2)
ChartLine(Quantity * Price / 2)
```



Note: The chart automatically rescales the axes to account for the range of the new data.

Setting Chart Axes

When you plot data in a chart, CA-Realizer automatically scales and labels the chart axes. The valid range and the data range of the plotted arrays determine the scale of chart axes.

The automatic scaling of axes, however, does not always produce the desired result. You may want to limit the range of plotted values, or you may want to force CA-Realizer to plot data according to a given scale.

CA-Realizer provides several commands for controlling chart axes. There are separate commands for changing the x- and y-axes—the x-axis commands apply to all panes in a chart, while the y-axis commands only affect the current pane.

The X-Axis

The x-axis is normally set according to the valid ranges of the arrays that are plotted. With the `ChartSetXAxis` command, you can change the range of indices shown on the x-axis. CA-Realizer only plots the data points with indices between the start and end of this range.

When the x-axis is based on numbers, `ChartSetXAxis` accepts two parameters to specify the starting and ending indices of a range:

```
ChartSetXAxis [(Start, End)]
```

When the x-axis has date values, `ChartSetXAxis` accepts additional parameters:

```
ChartSetXAxis(DateArray, Format, Interval1  
              [..., IntervalN ])
```

Setting the X-Axis with Date-Time Values

You can set the x-axis to date-time values instead of real values by using the second form of `ChartSetXAxis`:

```
ChartSetXAxis [(Times, Format, Interval1  
              [..., IntervalN ])]
```

Times should be an array of date-time values to use as the x-axis. The valid range of *Times* sets the range of the x-axis that will appear, and is used as the x-axis date correlation for plotting commands that specify a date array.

Format, which controls how the date-time values will be displayed on the axis, can be any date-time format string (see the `PRINT` command in the *CA-Realizer Language Reference Guide* for a list of all valid date-time formats).

Interval1 indicates the largest interval over which axis ticks should be placed. For example, if a tick is needed for every 10 entries in *Times*, *Interval1* should be set to 10. Or, if every entry in *Times* is an hour later than the previous, you can create daily ticks by setting *Interval1* to 24.

You can request any number of additional intervals for x-axis ticks, in decreasing size. For example, you can request ticks for every 30, 10, and 2 values with the following:

```
ChartSetXAxis (datevec, "D(M1/Y1)", 30, 10, 2)
```

If the interval specified provides enough screen space for the label to be printed, it will be. Otherwise, if there is room for a tick, the tick will be drawn.

Note: After you have used any `ChartSetXAxis` command, you can reset the x-axis back to its default based on all the data in the chart by using `ChartSetXAxis` without any parameters.

To use the second form of `ChartSetXAxis`, use *DateArray* as an array of date-time values as the x-axis. The valid range of *DateArray* sets the range of the x-axis that is displayed, and is used as the x-axis date correlation for plotting commands that specify a date array.

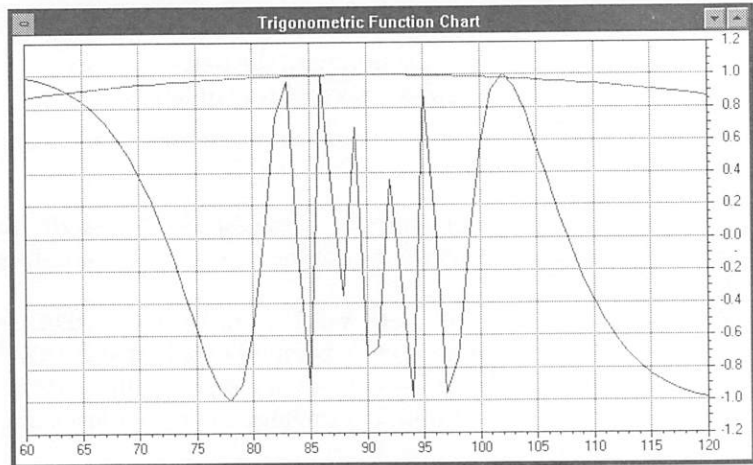
You can also use the `ChartSetXAxis` command to *zoom in* on a data set. Data points in a plot are displayed with greater detail and accuracy when you use a smaller range.

For example, modify the trigonometric function plot (created earlier in this chapter) as follows so that it displays values between 0 and 180 degrees:

```
' Manual\PG_11\TRIG2.RLZ

Radians = (3.141593/180) * Index(360)
SinTanWave = Sin(Tan(Radians))
SinWave = Sin(Radians)
Waves = ChartNew(, "Trigonometric Function Chart")
ChartSetColor(_Blue)
ChartLine(SinTanWave)
ChartSetColor(_Red)
ChartLine(SinWave)
ChartSelect(Waves)
ChartSetXAxis(60, 120)
ChartControl(_Show)
```

When you run this example, the following chart is displayed:



Restricting the range of the x-axis with `ChartSetXAxis` increases the scale of the data plot. CA-Realizer labels the x-axis from the starting index to the final index of the range. It ignores data values outside this range.

Setting a Secondary X-Axis

You can also display a secondary x-axis. This is used in conjunction with the date-time form of `ChartSetXAxis` to display a second row of labels:

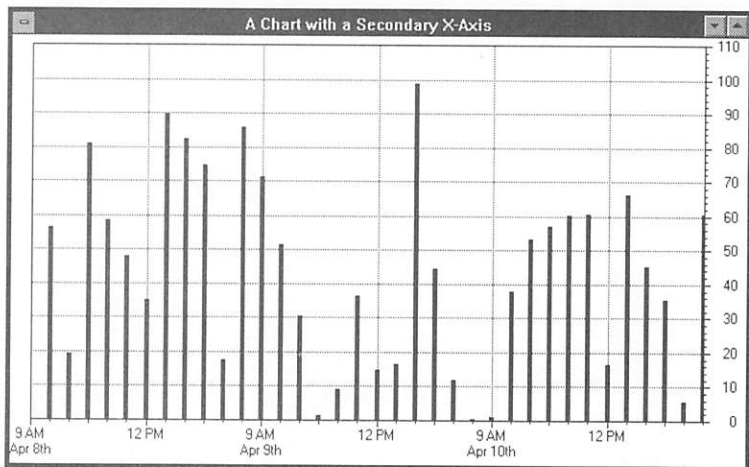
```
ChartSetXAxis2 [(Format, Interval1  
                [..., IntervalN])]
```

For example, with `ChartSetXAxis2`, the years can be labeled on the top part of the axis with the months shown below them.

`ChartSetXAxis2` is similar to `ChartSetXAxis`. It uses the same date vector specified by `ChartSetXAxis`, but has its own format string and its own interval values. If this command is called without parameters, the secondary x-axis is removed.

For example:

```
ChartNew( "A Chart with a Secondary X-Axis")  
Sales = Rnd(60) * 100  
Times = StrToDate("4/8/93 9:00 AM")  
Times = AddToTime(Times, 0, 30 * (Index(12)-1))  
Times = {Times, Times+1, Times+2}  
  
ChartBar(Sales; 4)  
ChartSetXAxis(Times, "D(h2 h5)", 12, 6, 1)  
ChartSetXAxis2("D(M3 D3)", 12)  
  
ChartControl(_Show)
```



Setting the View for the X-Axis

`ChartSetXAxis` sets the range of values that the x-axis represents. `ChartSetXView` sets the range of x-axis values shown at one time, enabling the user to zoom in on plotted data.

`ChartSetXView [(Beg, End)]`

If the range being viewed is less than the entire x-axis range, a horizontal scroll bar appears allowing the rest of the data to be viewed.

For example, suppose an array contains 1000 elements. Most likely, it will be difficult to view values of individual points in the array. By setting the x view to have a span of 50 points, only 50 points will be displayed at once. It will be much easier to distinguish individual points and the user can view all the data by scrolling through it.

If the range of values to view (*Beg* to *End*) is greater than the entire x-axis, or if it is outside the range of the chart, the scroll bar will not appear.

Beg and *End* can be either real values or date-time values. You can only use date-time values if you use the `ChartSetXAxis` command to set the x-axis with date-time values.

You can reset the view to the full size (without a scroll bar) by using `ChartSetXView` without any parameters.

Setting the Y-Axis

The `ChartSetYAxis` command sets the scale of the y-axis. The first two parameters of `ChartSetYAxis` specify the high and low range values for the axis.

For instance, adjust the trigonometric function chart with the following commands:

```
ChartSelect(Waves)  
ChartSetYAxis(-0.8 , 0.8)  
ChartControl(_Show)
```

If you invoke the `ChartSetYAxis` without parameters, CA-Realizer readjusts the y-axis automatically, according to the range of data values:

```
ChartSetYAxis  
ChartControl(_Show)
```

If *MinIncr* is given, labels are guaranteed to be separated by at least that value. If *MinIncr* is too small for the range of the y-axis, *IncrMethod* determines how the increments are adjusted to fit the data.

The table below lists the possible values for *IncrMethod*:

Value	Description
0	Does not adjust increments.
1	Multiplies by 2, 5, 10, 20, 50, 100, 200, and so on.
2	Multiplies by 2, 4, 8, 16, 32, and so on.
3	Uses endpoints exactly and fits the increments if possible.

If the values of *MinIncr* and *IncrMethod* make it difficult to create a reasonable axis, CA-Realizer will choose a default increment based upon the data.

If called without parameters, `ChartSetYAxis` resets the y-axis to its default based on the range of data in the pane.

Adding a Chart Key

When plotting more than one data set in a chart, you will often use a particular line color, line style, and marker style to distinguish one data set from another. In such cases, it is best to create a chart key. A chart key identifies which plotting style in a chart is used with which data set, as it contains fragments of the line and marker styles used in the chart, followed by descriptive text.

Adding Items to a Chart Key

You can build a chart key as you add data to the chart before or after the chart is complete. First, select the line or marker style you want to use for the particular data set, then use the `ChartSetKey` command. It adds an item to the key of the current chart:

```
ChartSetKey(Text [, Type] [; Font])
```

Text is the text for the new item, *Type* is the type of the graphic sample used for the item (either a line segment (the default), a marker, a small bar, or none), and *Font* specifies the font in which the new item should be displayed.

Note: An example of `ChartSetKey` is provided below.

Showing the Chart Key

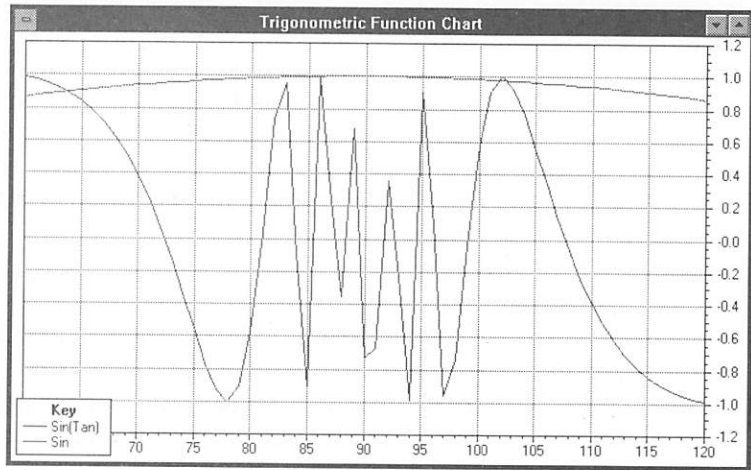
When you have finished adding items to a chart key, you can use the `ChartControlKey` command to display it.

For example, modify the trigonometric function plot (created earlier in this chapter) as follows to add a chart key:

```
' Manual\PG_11\TRIG3.RLZ  
  
Radians = (3.141593/180) * Index(360)  
SinTanWave = Sin(Tan(Radians))  
SinWave = Sin(Radians)  
Waves = ChartNew(, "Trigonometric Function Chart")  
ChartSetColor(_Blue)  
ChartLine(SinTanWave)
```

```
ChartSetColor(_Red)
ChartLine(SinWave)
ChartSelect(Waves)
ChartSetXAxis(60, 120)
ChartSelect(Waves)
ChartSetColor(_Blue)
ChartSetKey("Sin(Tan)", _Line)
ChartSetColor(_Red)
ChartSetKey("Sin", _Line)
ChartControlKey(_Show)
ChartControl(_Show)
```

The key appears in the bottom-left corner of the window:



You can reposition the key anywhere in the chart by clicking and dragging it with the mouse.

`ChartControlKey` affects the key in the currently selected chart. Additionally, similar to the other *TOOLControl* commands, you can use `ChartControlKey` to show, hide, size, and clear the chart key. (Sizing a chart key is demonstrated in the next section.)

Note: Unlike other CA-Realizer objects, you cannot specify the width and height of a chart key.

A Sample Chart Key

Here is another modification of the trigonometric chart:

```
' Manual\PG_11\TRIG4.RLZ

Radians = (_Pi/180) * Index(360)
SinTanWave = Sin(Tan(Radians))
SinWave = Sin(Radians)
Waves = ChartNew(, "Trigonometric Function Chart")
ChartSetXAxis(60,120)

ChartSetColor(_Red; _Data1)
ChartSetLineStyle(_Solid)
ChartSetKey("Sin(Tan) Line", _Line)
ChartLine(SinTanWave)

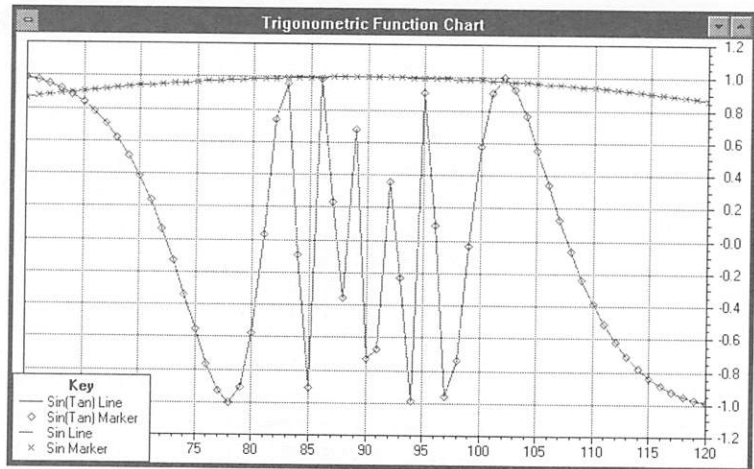
ChartSetColor(_Blue; _Data1)
ChartSetMark(_Diamond)
ChartSetKey("Sin(Tan) Marker", _Marker)
ChartMark(SinTanWave)

ChartSetColor(_Red; _Data1)
ChartSetLineStyle(_Dash)
ChartSetKey("Sin Line", _Line)
plotSinLine = ChartLine(SinWave)

ChartSetColor(_Blue; _Data1)
ChartSetMark(_XMark)
ChartSetKey("Sin Marker", _Marker)
plotSinMark = ChartMark(SinWave)

ChartControlKey(_Show)
ChartControl(_Show)
```

When you run this example, the following chart is displayed:

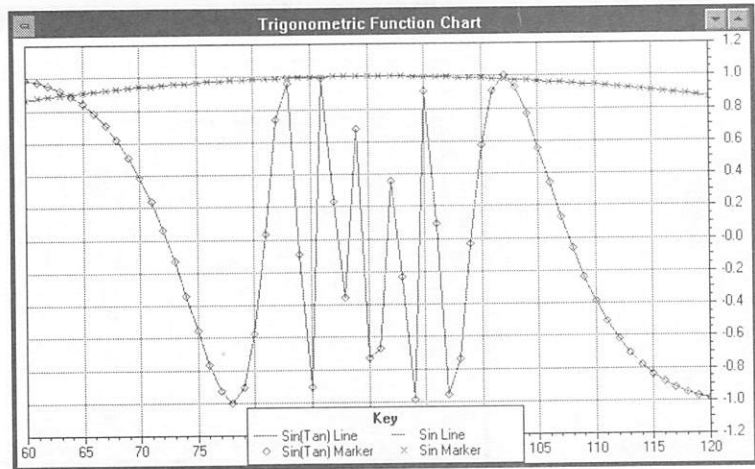


To some degree, the order of commands can be important when writing a CA-Realizer application. In the above example, notice that you set the characteristics of each plot (the color, the line style, or marker) before making a call to either `ChartLine` or `ChartMark`. Also notice that `ChartSetColor` sets the color for all of the chart commands until it encounters the next `ChartSetColor`. `ChartControlKey` and `ChartControl` are the last commands you use.

In addition to the form used above, `ChartControlKey` can take a size parameter with modifiers. This allows you to position a chart key anywhere in the chart. The first two modifiers determine the position of the key with respect to the upper-left corner of the chart. An optional third parameter specifies the number of columns in the key.

For instance, to move the current chart key and divide its contents into two columns, add the following statement to the previous example (*before* the `ChartControlKey(_Show)` statement):

```
ChartControlKey(_Size; _Center, _Bottom, 2)
```



Labeling Charts

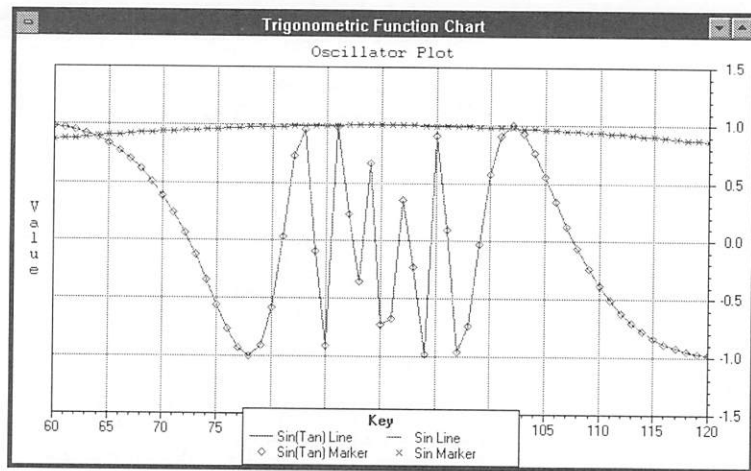
A complete chart includes labels and titles that describe the data displayed in the chart. You can place text along an axis and across the top of a chart, as well as labels next to data points.

Setting Chart and Axes Titles

You can place a title along the top, bottom, or sides of a chart with the `ChartSetTitle` command. Setting the title of a chart is different from setting the title of its window, since chart titles are contained within the charting area and you can control where they appear.

For example, to place some titles to the trigonometric chart created in the previous section, add the following statements just before the `ChartControlKey(_Show)` statement:

```
ChartSelect(Waves)
ChartSetTitle("Oscillator Plot", _Top)
ChartSetTitle("Degrees", _Bottom)
ChartSetTitle("Value", _Left)
```



Tip: You can also display the titles and grid lines of a chart in the same color (for example, `ChartSetColor(_Blue; _AllText)`).

Changing Axes Labels

Some charts require text labels in place of numbers on their x- and y-axes, such as some types of bar charts (for example, histograms) and some types of horizontal bar charts (like Gantt charts).

Labels are set with the `ChartSetXLabels` and `ChartSetYLabels` commands:

```
ChartSetXLabels(LabelsArray [, Elem [, RowNum]]
               [, _Stagger])
ChartSetYLabels(LabelsArray [, ElemNum])
```

ChartSetXLabels

Elem specifies the corresponding element. (By default, it starts at 1.) If *Elem* is a scalar and *Labels* is a vector, CA-Realizer places the labels for each element in order, starting with *Elem*.

RowNum can be 1, or 2, or an array containing 1's and 2's. (By default, *RowNum* is 1.) If you specify *_Stagger*, it arranges labels in two rows starting with *RowNum*.

ChartSetYLabels

ChartSetYLabels only affects the current pane. *ElemNum* specifies the corresponding element number. (By default, it starts at 1.) If *Elem* is a scalar and *Labels* is a vector, CA-Realizer places the labels for each element in order, starting with *Elem*. If *Elem* is a vector, its range must be the same as *Labels*.

Note: Both ChartSetXLabels and ChartSetYLabels without parameters remove labels and set the axis back to its default.

Hiding Labels

Chart titles and axes are visible by default. Use the ChartControlLabels command to hide the x-axis, y-axis, or the titles of the current pane. If you hide the labels and axes of a chart, CA-Realizer also hides the chart frame as well. In this case, the full chart area is used for plotting data.

ChartControlLabels affects the current pane. Here are some examples you can try with the trigonometric chart created in the previous section:

```
ChartSelect(Waves)
ChartControlLabels(_Hide, _XAxis + _YAxis)
ChartControlLabels(_Show, _Title)
```

Now try this:

```
ChartSelect(Waves)
ChartControlLabels(_Hide, _All)
```

You can restore hidden titles and axes with the ChartControlLabels(_Show, _All) command. Clear the title text using ChartControlLabels(_Clear, _Title).

Placing Text Within a Chart

Label a data point, bar, or line in a chart with the `ChartText` command. Use the following `ChartText` like any other plotting commands:

```
ChartText(XCoord, YCoord, Text)
```

```
ChartText(XCoord, YCoord, Text, _Screen  
          [, Position])
```

```
ChartText(XCoord, YCoord, Text, _Data [, Position])
```

The `_Screen` and `_Data` parameters determine which coordinate system (pane- or data-relative) to use for plotting text on the chart. (To position a data label relative to plotted data, use position values measured at the same scale as the axes. Data-relative coordinates are useful for labeling particular data points.)

Position indicates text alignment with respect to a particular coordinate point, and can be the sum of two of the following: `_CTPos_Left`, `_CTPos_Right`, `_CTPos_Top`, `_CTPos_Bottom`, and `_CTPos_Center`.

For instance, if *Position* is set to `_CTPos_Top + _CTPos_Left`, CA-Realizer positions the text so that the upper-left corner of the text is at (*XCoord*, *YCoord*).

Position can also be `_CTPos_Center` to center the text at the data point. In most instances, *Position* defaults to `_CTPos_Bottom + _CTPos_Left`.

Important! In CA-Realizer 1.0, the constants `_Left`, `_Right`, `_Top`, `_Bottom`, and `_Center` were allowed for the *Position* parameter. Those values are no longer allowed, and must be replaced with the corresponding `_CTPos` value.

Here is an example that marks the troughs and peaks in the previous trigonometric plot:

```
' Manual\PG_11\TRIG5.RLZ

Radians = (3.141593/180) * Index(360)
SinTanWave = Sin(Tan(Radians))
SinWave = Sin(Radians)
Waves = ChartNew(; "Trigonometric Function Chart")
ChartSetXAxis(60,120)

ChartSetColor(_Red; _Data1)
ChartSetLineStyle(_Solid)
ChartSetKey("Sin(Tan) Line", _Line)
ChartLine(SinTanWave)

ChartSetColor(_Blue; _Data1)
ChartSetMark(_Diamond)
ChartSetKey("Sin(Tan) Marker", _Marker)
ChartMark(SinTanWave)

ChartSetColor(_Red; _Data1)
ChartSetLineStyle(_Dash)
ChartSetKey("Sin Line", _Line)
plotSinLine = ChartLine(SinWave)

ChartSetColor(_Blue; _Data1)
ChartSetMark(_XMark)
ChartSetKey("Sin Marker", _Marker)
plotSinMark = ChartMark(SinWave)

ChartControlKey(_Show)

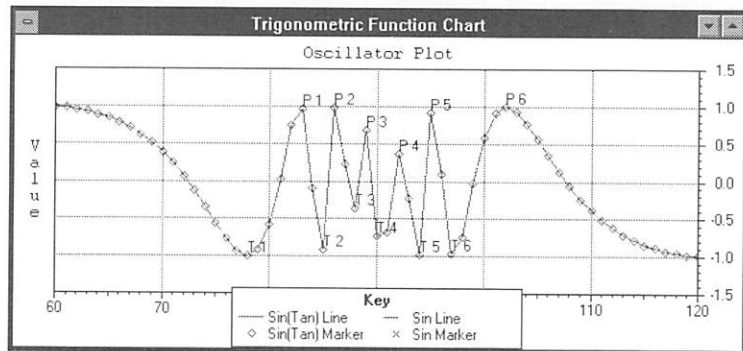
ChartControlKey(_Size; _Center, _Bottom, 2)

ChartSelect(Waves)
ChartSetTitle("Oscillator Plot", _Top)
ChartSetTitle("Degrees", _Bottom)
ChartSetTitle("Value", _Left)

CONST Rad = _Pi/180
ChartRemove(plotSinLine)
ChartRemove(plotSinMark)
XP = {83,86,89,92,95,102}
XT = {78,85,88,90,94,97}
PLabel = { "P 1", "P 2", "P 3", "P 4", \
           "P 5", "P 6"}
TLabel = {"T 1", "T 2", "T 3", "T 4", \
           "T 5", "T 6"}
FOR j = 1 TO EndValid(XP)
    YT[j] = SinTanWave[XT[j]]
    ChartText(XT[j], YT[j], TLabel[j], _Data)
    YP[j] = SinWave[XP[j]]
    ChartText(XP[j], YP[j], PLabel[j], _Data)
NEXT j
ChartControl(_Show)
```

Start by defining a constant value, *Rad*, which is used to convert degrees to radians. Next, use the `ChartRemove` command to remove the cosine plots so that the peaks and troughs you want to label are not obstructed. Then, define your x-coordinates and corresponding labels for both troughs and peaks. The `FOR` loop runs through the number of elements contained in one of the label arrays (the *XP* array) as determined by the `EndValid` command. This array contains the x values for each of the labels. This x value is also used as an index into the *SinTanWave* array to determine the y-coordinate. These two values are then passed to `ChartText` to plot the text.

Press `CTRL+ALT+F2` and run the above program to display the following chart:



Controlling the Chart Grid

By default, CA-Realizer draws a chart with a set of default horizontal and vertical grid lines. You can hide/show these grid lines, as well as customize them.

Hiding and Showing the Grid Lines

Use the `ChartControlGrid` command to alternately show and hide grid lines. For example, hide the vertical grid lines as follows:

```
ChartControlGrid(_Hide, _Vert)
```

Then hide the horizontal grid lines to remove the grid entirely:

```
ChartControlGrid(_Hide, _Horiz)
```

Now redisplay the entire grid:

```
ChartControlGrid(_Show, _All)
```

Creating a Custom Grid

Use the `ChartSetXGrid` and `ChartSetYGrid` commands to change the default grid settings. These commands allow you to replace the default x- and y-axis grids with custom grid lines.

```
ChartSetXGrid([Data [, Origin]  
              [, LineStyle][, Color])
```

```
ChartSetXGrid([ExactPoints]  
              [, LineStyle][, Color])
```

```
ChartSetYGrid([Data [, Origin]  
              [, LineStyle][, Color])
```

```
ChartSetYGrid([ExactPoints]  
              [, LineStyle][, Color])
```

Specify the location of both horizontal and vertical grids either as an *Origin* and *Data* between lines or as *ExactPoints* locations for each grid. (If *Data* and *Origin* or *ExactPoints* are not specified, default grids with the specified color and line style are used.)

If *Data* is a numeric scalar, it represents the interval between grid lines. If it is a numeric array, grid lines will be placed at the exact points specified by *Data*. If the x-axis was set with date-time values, *Data* for `ChartSetXAxis` can be an array of date-time values for the gridlines.

Additionally, *LineStyle* can be one of the following: `_Light` (dot-dot (the default)), `_Medium` (dash-dash), or `_Heavy` (solid), and *Color* can be any valid predefined or custom color. (The default color of chart grid is `_Gray`.)

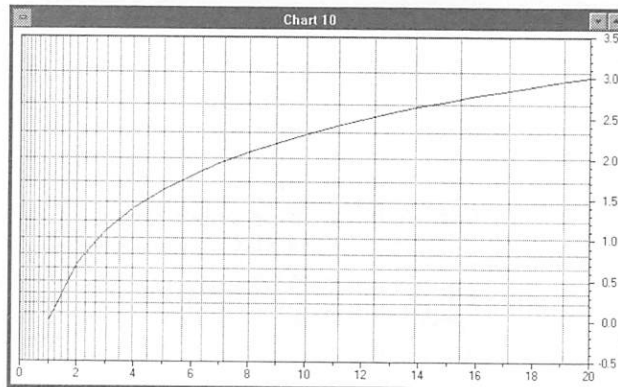
An unlimited number of grids can be defined for the current pane (note that all grids refer to the current pane). CA-Realizer paints grids in the same order as you define them. (If lines match, the latest one is on top.)

Existing grid lines are not removed until the grid is reset to its default. To reset a grid to its default, specify `ChartSetXGrid` and `ChartSetYGrid` without parameters or modifiers.

A Sample Custom Grid

Here is a simple example of `ChartSetXGrid` and `ChartSetYGrid` in use:

```
ChartNew(10)
ChartSetXGrid(Exp(Index(30)/10)-1)
ChartSetYGrid(Exp(Index(20)/10)-1)
ChartLine(Log(Index(20)))
ChartControl(_Show)
```



Setting Fonts for a Chart

CA-Realizer allows you to specify the fonts to be used for each x-axis (primary and secondary), each y-axis, each of the four titles, each string plotted with `ChartText`, the title of the key, and each entry in the key. Each of the elements in the chart that involve text can be created with a different font.

To set a font for one of these items, use `ChartSetFont`:

```
ChartSetFont (Font, Which)
```

Font is the font ID of an existing font. (If *Font* is 0, the default font for that element is used.)

Which identifies which chart element *Font* should be used with and can be one or a sum of the following constants:

Constant	Description
<code>_XAxis</code>	X-axis labels
<code>_XAxis2</code>	Secondary x-axis labels
<code>_YAxis</code>	Y-axis labels for the current pane
<code>_AllYAxis</code>	Y-axis labels for all panes
<code>_LeftTitle</code>	Left title
<code>_RightTitle</code>	Right title
<code>_TopTitle</code>	Top title
<code>_BottomTitle</code>	Bottom title
<code>_KeyTitle</code>	Key title
<code>_ChartText</code>	Text plotted with <code>ChartText</code> (affects any subsequent <code>ChartText</code> commands)
<code>_KeyText</code>	Text added to the key with <code>ChartSetKey</code> (affects any subsequent <code>ChartSetKey</code> commands)
<code>_All</code>	All elements of the chart

For example, let us modify the trigonometric plot chart created earlier in this chapter to change the fonts of the titles and the chart text labels using the `ChartSetFont` and `ChartText` commands.

This example defines some fonts (*fontTitle1*, *fontTitle2*, and *fontText*), and then inserts some `ChartSetFont` commands after setting the chart titles. Additionally, it also modifies the `ChartText` commands in the FOR loop to accept a font modifier (*fontText*).

Press CTRL+ALT+F2 and run the following program to display the results:

```
' Manual\PG_11\TRIG6.RLZ

Radians = (3.141593/180) * Index(360)
SinTanWave = Sin(Tan(Radians))
SinWave = Sin(Radians)
Waves = ChartNew( ; "Trigonometric Function Chart")
ChartSetXAxis(60,120)

ChartSetColor(_Red; _Data1)
ChartSetLineStyle(_Solid)
ChartSetKey("Sin(Tan) Line", _Line)
ChartLine(SinTanWave)

ChartSetColor(_Blue; _Data1)
ChartSetMark(_Diamond)
ChartSetKey("Sin(Tan) Marker", _Marker)
ChartMark(SinTanWave)

ChartSetColor(_Red; _Data1)
ChartSetLineStyle(_Dash)
ChartSetKey("Sin Line", _Line)
plotSinLine = ChartLine(SinWave)

ChartSetColor(_Blue; _Data1)
ChartSetMark(_XMark)
ChartSetKey("Sin Marker", _Marker)
plotSinMark = ChartMark(SinWave)

ChartControlKey(_Show)

ChartControlKey(_Size; _Center, _Bottom, 2)
```

```

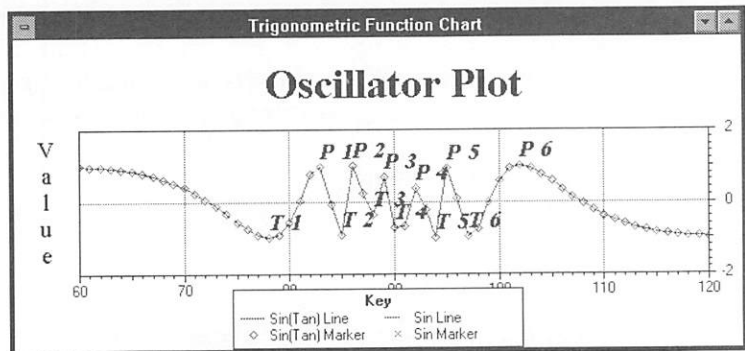
ChartSelect(Waves)
fontTitle1 = FontNew("Times New Roman", \
    24, _Bold)
fontTitle2 = FontNew("Times New Roman", 14)
fontText = FontNew("Times New Roman", \
    14, _Bold + _Italics)

ChartSetTitle("Oscillator Plot", _Top)
ChartSetFont(fontTitle1, _TopTitle)
ChartSetTitle("Degrees", _Bottom)
ChartSetFont(fontTitle2, _BottomTitle)
ChartSetTitle("Value", _Left)
ChartSetFont(fontTitle2, _LeftTitle)

CONST Rad = _Pi/180
ChartRemove(plotSinLine)
ChartRemove(plotSinMark)
XP = {83,86,89,92,95,102}
XT = {78,85,88,90,94,97}
PLabel = { "P 1", "P 2", "P 3", "P 4", \
    "P 5", "P 6"}
TLabel = {"T 1", "T 2", "T 3", "T 4", \
    "T 5", "T 6"}
FOR j = 1 TO EndValid(XP)
    YT[j] = SinTanWave[XT[j]]
    ChartText(XT[j], YT[j], TLabel[j], _Data; \
        fontText)
    YP[j] = SinTanWave[XP[j]]
    ChartText(XP[j], YP[j], PLabel[j], _Data; \
        fontText)
NEXT j

ChartControl(_Show)

```



Advanced Charting Features

There are additional advanced charting features that extend the flexibility of the chart tool even further. These features include:

- Updating plots
- Removing plots
- Querying plot values
- Creating multiple panes
- Redrawing charts

Updating a Plot

Once a plot has been added to a chart, it can be updated in a variety of ways using the `ChartUpdate` command. For example, new elements can be added to the end of the plot (as with stock market data that is being plotted “live”), elements within the plot can be modified, or all the data associated with the plot can be changed.

The syntax for the `ChartUpdate` command depends on the chart command that was used to create the plot. In each case, the first parameter is the unique plot number (*PlotNum*) that was returned by one of the functions that created the chart plot, the first modifier is the type of update (*UpdateType*), and the remaining parameters are the new or updated data values.

For example:

```
'Create a chart
ChartNew(10)
.
.
.
'Set x- and y-axis values
.
.
.
'Add new data and update the plot
a = ChartLine(Rnd(10))
FOR J = 1 TO 90
    ChartUpdate(a, Rnd(1); _Append)
NEXT J
```

Removing a Plot

You can remove a plot from a chart with the `ChartRemove` command. It requires that you specify the unique number associated with the plot to be removed. (This number was returned by one of the functions that created the chart plot.)

This example creates a chart with two plots (1 and 2) and then removes one of them:

```
'Create two plots
ChartNew(1)
plot1 = ChartLine(Index(20))
plot2 = ChartBar(Rnd(15);10)

'Remove the line chart
ChartRemove(plot1)
```

Querying Axis Values

You can query the plot values on the x- and y-axes using the `ChartQXAxis` and `ChartQYAxis` commands.

`ChartQXAxis`

The `ChartQXAxis` command allows the user to query the chart and find out the actual range of values being used for the axes.

```
Range = ChartQXAxis
```

It returns a two-element array with the first and last values shown on the x-axis, even if `ChartSetXView` has been used. The values will be real, unless a date-time array was set with `ChartSetXAxis`.

`ChartQYAxis`

This command returns a two-element array with the first and last values shown on the y-axis for the current pane.

```
Range = ChartQYAxis
```

Creating Multiple Panes

Panes enable plots with different scales to be superimposed and also allow charts to be broken into several distinct regions.

Panes are bands that stretch across the complete width of a chart. There is a current pane (just as there is a current chart) and all chart commands affect the current pane. Additionally, each pane can have its own y-axis.

The number of panes is not fixed, and new panes can be created at any time by simply selecting them.

To superimpose several plots with very different ranges, create several panes within the same region of the chart. You may have to use `ChartSetYAxis` to ensure that the axes are identical. This command sets the range and label format for the y-axis of the current chart pane.

Panes are also useful for breaking a chart into distinct regions. For instance, trading volume and stock prices may need to be graphed in the same chart. A pane created at the bottom of this chart may contain a bar chart of volume, and another pane at the top of the chart may contain a high-low-close graph of the prices. Both panes share the same x-axis, but each has its own y-axis.

The following commands are used to manipulate panes (they are demonstrated in the sample program that follows):

`ChartSelectPane`
(*Pane*)

You can change the current pane with the `ChartSelectPane` command as follows:

```
ChartSelectPane(Pane)
```

Pane is the number of the pane to select. If the pane does not already exist, it will be created automatically.

ChartControlPane (_Size)

Panes always take up the entire width of the chart. You can adjust the position and height of the current pane with `ChartControlPane(_Size)`.

```
ChartControlPane(_Size [; Top, Height [, AxisLoc]])
```

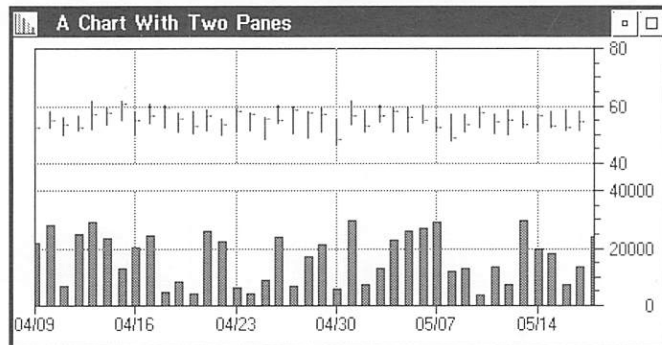
Using the CA-Realizer position notation, *Top* specifies the location of the top of the pane and *Height* specifies the height of the pane. *AxisLoc* specifies the location of the y-axis labels, either `_Left`, `_Right`, or `_None`.

ChartControlPane (_Clear)

Use the `ChartControlPane(_Clear)` command to clear the plots and text in the current pane. Plots in other panes will remain unaffected.

The above commands are used in the following program to create a multiple pane chart using random data:

```
ChartNew("A Chart With Two Panes")
Dates = StrToDate("04/08/91") + Index(40)
ChartSetXAxis(Dates, "D(M1/D1)", 7)
Volume = Rnd(40) * 30000
High = 55 + Rnd(40) * 7
Low = High - Rnd(40) * 5 - 5
Close = Low + (High - Low) * Rnd(40)
ChartSelectPane(1)
ChartControlPane(_Size; 0 pct, 45 pct)
ChartBar(High, Low, Close)
ChartSelectPane(2)
ChartControlPane(_Size; 55 pct, 45 pct)
ChartSetColor(_Red)
ChartBar(Volume; 5)
ChartControl(_Show)
```



Controlling the Redraw

There are various controls that allow the chart to be redrawn.

ChartControl
(_SetRedraw)

You can use ChartControl(_SetRedraw) to specify whether new plots and chart commands take effect immediately, or are accumulated for a single update and redraw. Turning the redraw off is very useful if a number of complex modifications are made to the chart at once, since the intermediate results are not shown, and the entire operation is faster

```
ChartControl(_SetRedraw [; [StyleFlags,] [Mod1,  
Mod2 [, Mod3, Mod4]]])
```

ChartControl(_SetRedraw; 0) turns the redraw off.

ChartControl(_SetRedraw; 1) turns the redraw back on and refreshes the chart.

ChartControl(_Show)

When the redraw is off, ChartControl(_Show) can also be used to update the display.

There are a few user actions that will cause the chart to be redrawn even when the redraw is off, including changes in the chart size or scrolling when ChartSetXView is active.

ChartQ(_Redraw)

ChartQ(_Redraw) returns the current redraw status for the chart.

Setting a Chart Procedure

When the user clicks in a chart or tries to close it, the chart's procedure is invoked. You can use a chart procedure to process these clicks in different ways. For example, your application might zoom into the selected region to show more detail or it might force the chart's key to appear or disappear. You could also use clicking on a data point to bring up another chart or spreadsheet with information related to that value.

Chart procedures are very powerful and you use them to customize charting applications.

Tip: See the "Overview of the CA-Realizer Programmable Application Tools" chapter for information about tool procedures.

Attaching a Chart Procedure

Use `ChartSetProc` to associate a procedure or program module with the current chart:

```
ChartSetProc[(ProcName | ModuleName [: _Size])]
```

ProcName should be the name of a defined procedure that takes one parameter (the tool procedure parameter array).

ModuleName is the name of a file to run.

If the `_Size` modifier is used, the chart procedure will also be called when the user resizes the chart window.

Note: `ChartSetProc` without any parameters disassociates the current chart procedure from the chart.

Processing in the Chart Procedure

The basic structure of a chart procedure is as follows:

```
PROC MyChartProc(params)
  ChartSelect(params[_ItemNum], params[_FormNum])
  SELECT CASE params[_Invoke]
    CASE _Close
      'Possibly do something when the chart
      'is closing. To prevent the close, do:
      'params[_UseRealizer] = 0

    CASE _Click
      'Process the mouse click here.
      'params[_XPos] and params[_YPos] are
      'the point in the chart where the
      'mouse was clicked, specified in
      'pixels. Use ChartQPtInfo to convert
      'it to data relative coordinates.
  END SELECT
END PROC
```

Remember, you can always prevent the chart from closing by setting the `_UseRealizer` element of the chart procedure parameter array to 0.

Converting Screen-Relative to Data-Relative Positions

`ChartQPtInfo` converts a screen-relative position to a data-relative position. It is typically used after a mouse click in a chart to correlate a pixel position with a particular data point.

```
ReturnArray = ChartQPtInfo(X, Y [, Pane ])
```

When the user clicks in the chart to invoke the chart procedure, the `_XPos` and `_YPos` elements of the chart procedure parameter array are set to the location of the click. This location is measured in pixels, relative to the upper-left corner of the chart. Use `ChartQPtInfo` to convert these values to data-relative values.

You most often use the `ChartQPtInfo` function as follows:

```
point = ChartQPtInfo(params[_XPos], params[_YPos])
```

`ChartQPtInfo` returns a three-element array. The first two elements are the x- and y-data coordinates of the point, relative to the x- and y-axes for the pane. (If the x-axis is set with a date-time array, the x value is the index into that array.)

The fractional portion indicates how close the point is to the next element. If the x value is 3.2, the clicked point is two-tenths of the way between elements 3 and 4 of the array.

The third element of the array is the number of the pane containing the point. If several panes overlap and you do not specify the *Pane* parameter, CA-Realizer uses the lowest numbered pane. If you specify *Pane* and it is among the overlapping panes, the third element will be *Pane*. (If there is no data in the pane, or if the clicked point is not within any pane, all three elements in the returned array will be set to 0.)

A Sample Chart Procedure

Here is a useful example of a chart procedure. It draws three data lines in three different colors. When the user clicks in the chart, the chart procedure figures out which plot the click was closest to, and then places a marker at the spot using the color of that plot.

This program uses a number of interesting techniques, including arrays of arrays and FirstMatch. The closest point algorithm is not perfect, however—it only finds the nearest distance in the y direction. A better algorithm would compute the perpendicular distance from the point to the line.

```
' Manual\PG_11\NEARLINE.RLZ

PROC ChartProc(params)
  'Get the clicked point in data values
  point = ChartQPtInfo(params[_XPos], \
    params[_YPos])
  'Find the distance to each line
  FOR i = 1 TO NumPlots
    y1 = Data[i][Floor(point[1])]
    y2 = Data[i][Ceil(point[1])]
    fract = point[1] MOD 1.0
    y = y1 + fract * (y2-y1)
    dy[i] = Abs(y - point[2])
  NEXT i
  'Determine the shortest distance
  match = FirstMatch(dy, Min(dy))

  'Place a mark at the point
  ChartSetColor(Colors[match])
  ChartXYMark({point[1]}, {point[2]})
END PROC

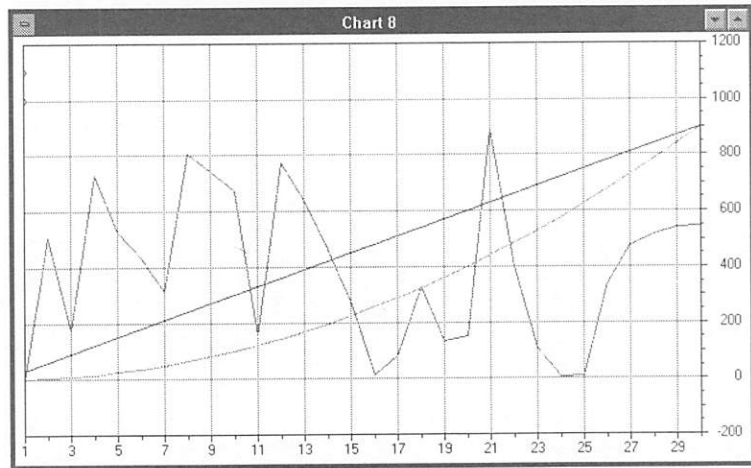
NumPlots = 3
ChartNew

'Create three data plots
Data[1] = Rnd(30) * 900
Data[2] = Sqr(Index(30))
Data[3] = Index(30) * 30
Colors = {_Red, _Green, _Blue}

'Plot the arrays
FOR i = 1 TO NumPlots
  ChartSetColor(Colors[i])
  ChartLine(Data[i])
NEXT i

'Display the chart and set the procedure

ChartSetXAxis(1, 30)
ChartSetProc(ChartProc)
ChartControl(_Show)
```



Printing a Chart

To print the current chart, use `ChartControl(_Print)`. This command places an image of the chart in a temporary output file held by the Windows or OS/2 Print Manager. This file continues to collect output until it is closed with `LFLUSH`.

CA-Realizer prints charts exactly the way they appear in the chart window. When printing a chart on a black and white printer, it converts color to a grayscale approximation. To produce the clearest printed output, plot data on a white background. In general, high contrast colors look best.

Use the following commands to print some of the charts created in this chapter:

```
ChartSelect (Waves)
ChartControl (_Print)
LFLUSH
```


Chapter 12

Spreadsheets

In This Chapter	12-1
The Anatomy of a Spreadsheet	12-2
Creating a New Spreadsheet	12-3
Using SheetNew	12-3
Spreadsheet Data	12-4
Controlling a Spreadsheet	12-6
Setting Fonts for Cells and Headings	12-8
Setting the Fonts	12-8
Adjusting the Cell Width	12-9
Using Column Buttons	12-10
Restricting Cells, Rows, and Columns	12-11
Setting a Range	12-11
Setting a Read-Only Attribute	12-11
Example	12-12
Setting the Default Format	12-12
Moving to a Cell	12-13
Controlling Row and Column Headers	12-14
Changing Row Headers	12-14
Changing Column Headers	12-15
Hiding Row and Column Headers	12-16
Updating the Contents of a Spreadsheet	12-17
Adding New Variables	12-17
Recalculating the Data	12-18
Removing Cleared Data	12-18
Example	12-19

Querying the Spreadsheet Selection	12-20
Setting a Spreadsheet Procedure	12-22
Attaching a Spreadsheet Procedure	12-22
Processing in the Spreadsheet Procedure	12-23
A Sample Spreadsheet Procedure	12-24

Chapter 12

Spreadsheets

In This Chapter

This chapter describes the CA-Realizer spreadsheet tool. Spreadsheets provide an easy way to view and manipulate variables, as they present variables in a tabular format, with each variable or family member as a column in the spreadsheet. The user can view data, scroll through the spreadsheet, and edit the values in the cells.

Note: The commands you use to create and manipulate a spreadsheet—such as SheetNew, SheetQUnique, SheetControl, and SheetSelect—work like their other tool counterparts. See the “Overview of the CA-Realizer Programmable Application Tools” chapter for details about the standard tool commands.

The following table briefly describes the commands and functions that can be used to create and control spreadsheets:

Command	Description
SheetControl	Allows you to control the attributes of the current sheet.
SheetControlLabels	Allows you to control the row and column headers in the current sheet.
SheetNew	Creates a new sheet.
SheetQ	Returns information about a sheet.
SheetQInfo	Returns information about the rows and columns in a sheet.

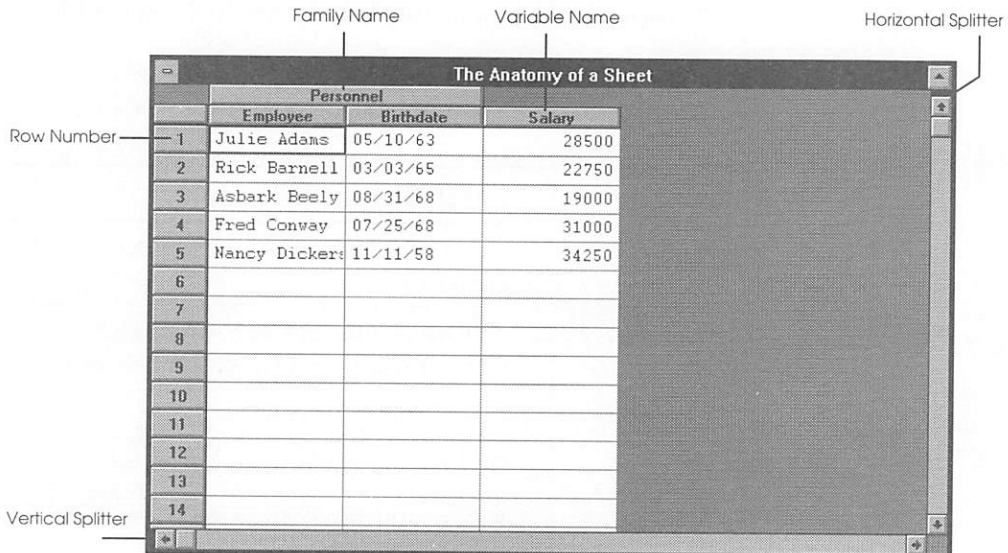
Continued

Continued

Command	Description
SheetQUnique	Returns a unique sheet number.
SheetSelect	Selects a sheet to be the current sheet.
SheetSetColLabels	Sets labels for columns.
SheetSetProc	Assigns a procedure to the current sheet.
SheetSetRowLabels	Sets labels for rows.
SheetUpdate	Adds and updates variables in the current sheet.

The Anatomy of a Spreadsheet

A spreadsheet is comprised of many parts, most of which can be controlled separately (as discussed in this chapter). This graphic outlines some basic terms associated with spreadsheets:



Creating a New Spreadsheet

Like any CA-Realizer tool, each sheet must have a unique ID. You can use the `SheetNew` or `SheetQUnique` command to generate one, or you can specify a particular ID when issuing the `SheetNew` command.

Using SheetNew

The general form of the `SheetNew` command is as follows:

```
[SheetID = ] SheetNew(SheetNum, [Data1
    [ ..., DataN]] [; Title [, Style]])
```

Variables can be placed in the sheet when it is created by specifying them as *Data1* to *DataN*. (Other variables can be added after the sheet is created by using the `SheetUpdate` command.)

Note: If *Data1* is a two-dimensional array, no other variables can be added to the sheet.

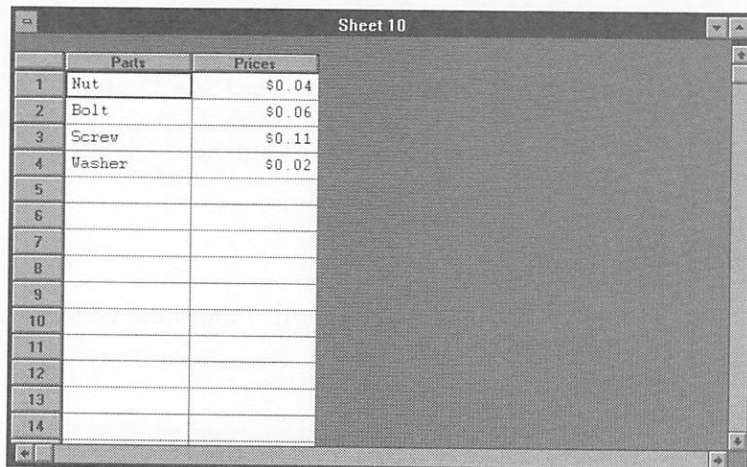
Use *Title* and *Style* to override the sheet's default attributes. The sheet's title is initially set to "Sheet *SheetNum*" (for example, Sheet 10). Specifying *Title* allows you to set it to a different text string.

Additionally, the default window style of a sheet is `_Title + _Size + _Minimize + _Close`. Using the *Style* parameter, you can add additional styles or remove some or all default ones. (Available *Style* constants are listed in the *Language Reference Guide*.)

For example:

```
'Create a new sheet with part data
Parts = {"Nut", "Bolt", "Screw", "Washer"}
Prices = {0.04, 0.06, 0.11, 0.02}
SetFormat(Prices, "$$.##")
SheetID = SheetQUnique
SheetNew(SheetID, Parts, Prices)
SheetControl(_Show)
```

Running this program displays the following:



	Parts	Prices
1	Nut	\$0.04
2	Bolt	\$0.06
3	Screw	\$0.11
4	Washer	\$0.02
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		

Note: When a sheet is created, it is invisible. SheetControl can be used to make it visible; SetFormat can be used to change the format of a column; and SheetUpdate can be used to adjust the width of a column.

Spreadsheet Data

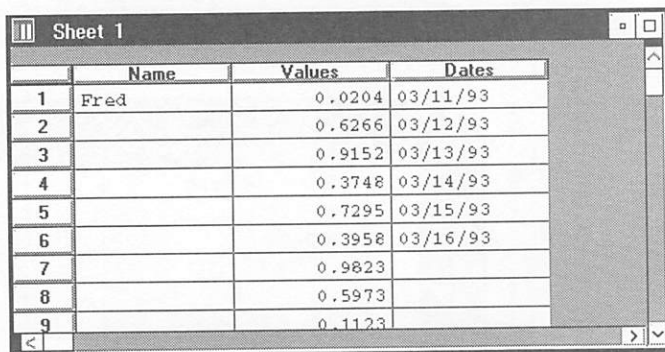
The data in the sheet is always a view of one or more CA-Realizer variables. These values can be:

- Scalars
Use an entire column but appear only in the first row.
- One-dimensional arrays
Fill a column and can be expanded with additional rows and elements.
- Families
Display a second-level column header with the family name and use a column for each family member.
- Two-dimensional arrays
Use the entire spreadsheet, and can expand both in the number of rows and columns.

Scalars and Arrays

Here is an example of scalars and arrays:

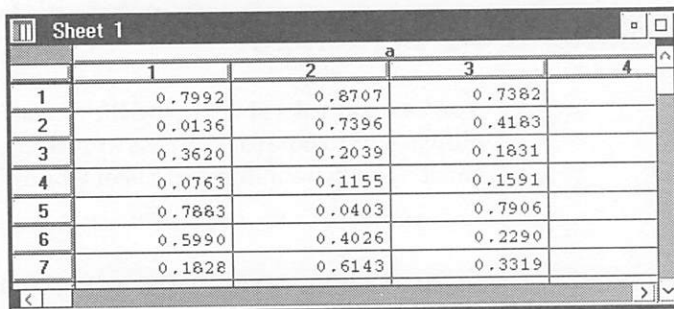
```
Name = "Fred"
Values = Rnd(10)
Dates = StrToDate("3/10/93") + Index(6)
SheetNew(1, Name, Values, Dates)
SheetControl(_Show)
```



	Name	Values	Dates
1	Fred	0.0204	03/11/93
2		0.6266	03/12/93
3		0.9152	03/13/93
4		0.3748	03/14/93
5		0.7295	03/15/93
6		0.3958	03/16/93
7		0.9823	
8		0.5973	
9		0.1123	

When two-dimensional arrays are used, they take over the entire spreadsheet to allow expansion in both directions:

```
a[1:7, 1:3] = Rnd(21)
SheetNew(1, a)
SheetControl(_Show)
```

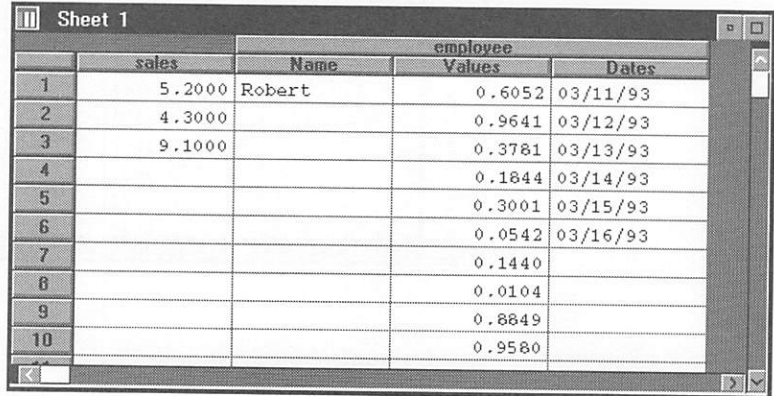


	1	2	3	4
1	0.7992	0.8707	0.7382	
2	0.0136	0.7396	0.4183	
3	0.3620	0.2039	0.1831	
4	0.0763	0.1155	0.1591	
5	0.7883	0.0403	0.7906	
6	0.5990	0.4026	0.2290	
7	0.1828	0.6143	0.3319	

Families

Here is an example involving a family:

```
employee.name = "Robert"
employee.values = Rnd(10)
employee.dates = StrToDate("3/10/93") + Index(6)
sales = {5.2, 4.3, 9.1}
SheetNew(1, sales, employee)
SheetControl(_Show)
```



	sales	employee Name	employee Values	Dates
1	5.2000	Robert	0.6052	03/11/93
2	4.3000		0.9641	03/12/93
3	9.1000		0.3781	03/13/93
4			0.1844	03/14/93
5			0.3001	03/15/93
6			0.0542	03/16/93
7			0.1440	
8			0.0104	
9			0.8849	
10			0.9580	

Controlling a Spreadsheet

You can control a sheet using the `SheetControl` command. It allows you to control both the appearance and the behavior of a sheet. The general form of `Sheet Control` is:

```
SheetControl(Action [; Mod1 [,Mod2 [,Mod3, Mod4]]])
```

Action can be one of a number of constants (they are listed in the table below). The values you should supply for *Mod1* through *Mod4* depend on the constant supplied for *Action*.

All of the standard *Action* values that are common to all the tools—such as `_Size`, `_Show`, and `_Minimize`—can be used. (See the “Overview of the CA-Realizer Programmable Application Tools” chapter for details.)

The spreadsheet-specific *Action* constants are listed below:

Constant	Description
<code>_ColButton</code>	Determines how a sheet will react when the user presses the column heading button. With <code>_ColButton</code> , <i>Mod1</i> can be <code>_Select</code> or <code>_Format</code> . <code>_Select</code> —the default—selects the entire column; <code>_Format</code> displays the format selection dialog box.
<code>_ColRange</code>	Sets the range of a column. <i>Mod1</i> sets the lower boundary and <i>Mod2</i> the upper boundary.
<code>_ReadOnly</code>	Determines how the cells in a sheet can be used. If <i>Mod1</i> is 1, the sheet cells are read only. If <i>Mod1</i> is 0, the data displayed in a sheet's cells can be edited by the user.
<code>_RowRange</code>	Sets the range of a row. <i>Mod1</i> sets the lower boundary and <i>Mod2</i> the upper boundary.
<code>_Scroll</code>	Scrolls to a cell in a specified row. <i>Mod1</i> is the number of the row, and <i>Mod2</i> is either the variable or the number of the column.
<code>_SetColor</code>	Sets a color to be used for cells and row and column headings. (The default color for these items is <code>_Black</code> .) <i>Mod1</i> sets the cell color, and <i>Mod2</i> the row and column header colors.
<code>_SetFont</code>	Sets a font to be used for cells and row and column headings. <i>Mod1</i> sets the cell font, and <i>Mod2</i> the row and column header fonts.

Note: By default, when you create a sheet, it becomes the current sheet. This means that all subsequent sheet commands apply to it. To set a different sheet to be the current sheet, use `SheetSelect`.

Setting Fonts for Cells and Headings

You can specify a font (or fonts) in which to display a sheet's cells and row and column headings using the `SheetControl` command with the `_SetFont` parameter. Note that:

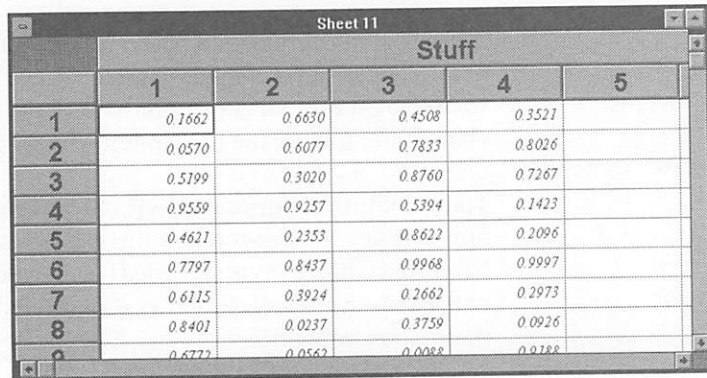
- All cells and headings use the specified font.
- The cell font controls the height of the cells.
- A cell's width can be modified using the `SheetUpdate` command.

Setting the Fonts

When using `SheetControl(_SetFont)`, specify the font ID of the font to be set for the sheet's cells as *Mod1*, and use *Mod2* to specify the font ID to be used for the sheet's row and column headers. For example, the following creates two fonts—Helvetica, 18 point, bold and Times New Roman, 10 point, italic—and uses them to format the sheet's row and column headings and cells:

```
Stuff[1:10,1:4] = Rnd(40)
MySheet = SheetNew(0, Stuff)
Font1 = FontQUnique
FontNew(Font1; "Helvetica", 18, _Bold)
Font2 = FontQUnique
FontNew(Font2; "Times New Roman", 10, _Italics)
SheetControl(_SetFont; Font2, Font1)
SheetControl(_Size; _Right, _Center, 80 pct, 60 pct)
SheetControl(_Show)
```

The following sheet is created:



	Stuff				
	1	2	3	4	5
1	0.1662	0.6630	0.4508	0.3521	
2	0.0570	0.6077	0.7833	0.8026	
3	0.5199	0.3020	0.8760	0.7267	
4	0.9559	0.9257	0.5394	0.1423	
5	0.4621	0.2353	0.8622	0.2096	
6	0.7797	0.8437	0.9968	0.9997	
7	0.6115	0.3924	0.2662	0.2973	
8	0.8401	0.0237	0.3759	0.0926	
9	0.6772	0.0562	0.0088	0.0188	

Note: To restore a font to its original default value, set *Mod1* or *Mod2* to 0.

Adjusting the Cell Width

When you create a spreadsheet, all columns in the sheet have the same default width of 12 characters. Though this width is often sufficient for displaying most values, others may require more than 12 characters.

One way to adjust the column width to fit your data is to resize the column with the mouse. Place the mouse pointer near the edge of the column label. When the pointer changes to the column-resizing cursor, drag the mouse to change the size of the column. This method, however, may not always be ideal, because you must adjust the column width every time you create the spreadsheet.

Another way to adjust column widths is to use the `SheetUpdate` command to specify the width of a column while you are adding to or updating the sheet's variables. (This is discussed later in this chapter.)

Using Column Buttons

When the user chooses a column button (the column header), you can instruct the sheet to either select the entire column, or to display a Format selection dialog box, which lets the user change the format for the contents of the entire column.

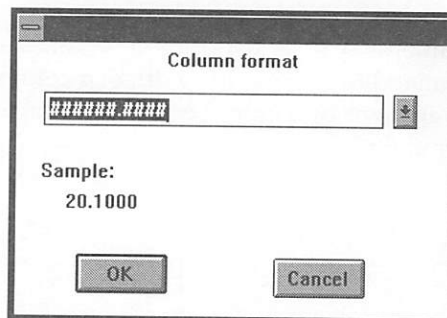
The `_ColButton` parameter with `SheetControl` lets you control how a sheet will react when the user presses the column heading button. If you set `Mod1` to be `_Select` (the default), the entire column is selected; if you set `Mod1` to be `_Format`, the dialog box appears.

For example, try running the following program and then choosing the column header for *HighPrice*:

```
Font1 = FontQUnique
FontNew(Font1; "Helvetica", 14, _Bold)
Font2 = FontQUnique
FontNew(Font2; "Times New Roman", 10)
FileImport("MANUAL\STOCK.DAT", _Realizer, _Named)

StockSheet = SheetQUnique
SheetNew(StockSheet, OpenPrice, HighPrice, \
        LowPrice, ClosePrice; "Stock Data")
SheetControl(_Size; _Right, _Center, 80 pct, 60 pct)
SheetControl(_ColButton; _Select)
SheetControl(_SetFont; Font2, Font1)
SheetControl(_SetColor; _Red, _Blue)
SheetControl(_Show)
```

Now change the `_Select` parameter of `SheetControl` to `_Format`, and choose the same column header to compare the difference. You will see a dialog box, similar to the one below, that allows you to either choose from a list of standard formats or enter your own:



Restricting Cells, Rows, and Columns

By default, sheets allow the user to scroll past the range of data that is in a column. In addition, sheets allow users to extend the data by entering new values—if the user changes the data in the spreadsheet, the actual values in the variables and arrays used to populate the sheet are also changed. (The same is true for the columns of a two-dimensional array.) Likewise, if the user adds elements, the corresponding arrays will be expanded.

Setting a Range

You can restrict the ranges of values in a sheet that are visible and accessible to a user with the `SheetControl(_RowRange)` and `SheetControl(_ColRange)` commands. They prevent the user from expanding beyond the desired range.

Note: Also, the sheet procedure can be invoked when cell changes occur to allow the program to process the changes and update other aspects of the display accordingly. This is discussed later in this chapter.

For the `_RowRange` and `_ColRange` `SheetControl Action` constants, *Mod1* and *Mod2* are the minimum and maximum row or column numbers that will be visible. When set, the user will not be able to scroll beyond that range.

Note: An example is provided later in this section.

Setting a Read-Only Attribute

You can use the `SheetControl _ReadOnly` constant to prevent the user from modifying or extending the data in a sheet. To make a sheet read-only, set *Mod1* to 1; to allow changes again, set it to 0.

Example

This program creates a sheet using the elements in the *a* array. It restricts the user from scrolling before or past rows 3 and 9 and columns 1 and 4, and also prevents the user from editing the sheet:

```
a[1:10, 1:10] = Rnd(100)
SheetNew(1, a)
SheetControl(_RowRange; 3, 9)
SheetControl(_ColRange; 1, 4)
SheetControl(_ReadOnly; 1)
SheetControl(_Show)
```

	a			
	1	2	3	4
3	0.8230	0.1519	0.6255	0.3147
4	0.9315	0.8699	0.8665	0.6745
5	0.4159	0.4635	0.9792	0.1264
6	0.9568	0.0281	0.3187	0.7570
7	0.4419	0.9150	0.5723	0.1188
8	0.8730	0.4270	0.3582	0.3820
9	0.7734	0.2448	0.3428	0.2300

Setting the Default Format

How the data in a sheet is displayed depends on the format defined for the variables that are associated with that sheet data. Each variable is of a specific data type; in turn, each data type has a default format string. (A *format string* describes how a value looks when it is displayed.) When a log or spreadsheet displays values, they appear in their default format.

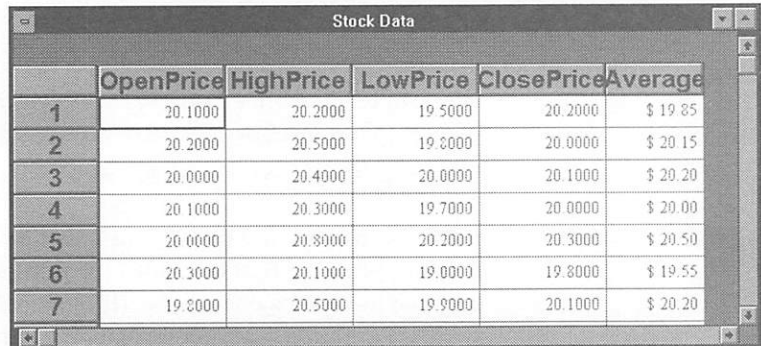
Note: Default format strings (as well as all available format strings) are listed in the PRINT command in the *CA-Realizer Language Reference Guide*.

You can, however, format the contents of a spreadsheet. Simply execute the SetFormat command with two parameters—the first is the name of the variable for which a new format should be set and the second is a string that holds the new format string.

The next time the variable is displayed in a spreadsheet (or with a PRINT command), it will appear in its new format.

For example, modify the previous code to include the following:

```
Average = (HighPrice + LowPrice) / 2
StockSheet = SheetQUnique
SetFormat(Average, "$###.##")
SheetUpdate(Average)
```



	OpenPrice	HighPrice	LowPrice	ClosePrice	Average
1	20.1000	20.2000	19.5000	20.2000	\$ 19.85
2	20.2000	20.5000	19.8000	20.0000	\$ 20.15
3	20.0000	20.4000	20.0000	20.1000	\$ 20.20
4	20.1000	20.3000	19.7000	20.0000	\$ 20.00
5	20.0000	20.8000	20.2000	20.3000	\$ 20.50
6	20.3000	20.1000	19.0000	19.8000	\$ 19.55
7	19.8000	20.5000	19.9000	20.1000	\$ 20.20

Notice that *Average* is formatted into dollars and cents, and the column width has been automatically adjusted by the *SheetUpdate* command. When *SheetUpdate* is called with a new variable name, CA-Realizer adds a column for that variable to the spreadsheet.

Moving to a Cell

When a spreadsheet is larger than its window, not every spreadsheet cell is displayed. In this situation, you can use the scroll bars to see other cells of a spreadsheet.

You can also instruct CA-Realizer to automatically scroll a spreadsheet to reveal a particular cell with the *_Scroll* constant of the *SheetControl* command.

Two modifiers indicate the column name and row number of a cell. For example, to scroll to the cell in row 15 of the *Average* variable column in the example above, use the following command:

```
SheetControl(_Scroll; Average, 15)
```

Controlling Row and Column Headers

By default, the rows in a sheet are labeled with row numbers and the columns are labeled with the variable names (and, if applicable, a family header). These default labels can be changed to custom text, or can be hidden.

Changing Row Headers

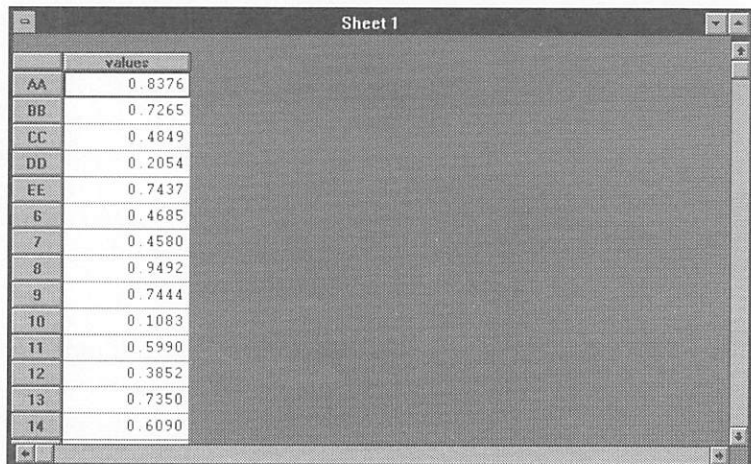
You can use the `SheetSetRowLabels` command to replace row headers with a set of specified labels as follows:

```
SheetSetRowLabels [(Labels)]
```

Labels is an array of labels, which will be assigned to all rows starting with the first one until the end of the labels. Repeated usage will replace old labels. (If `SheetSetRowLabels` is called without the *Labels* parameter, labels are restored to their original default values.)

The following code changes the first five row headers in a spreadsheet from randomly-generated numbers to letters:

```
values = Rnd(20)
SheetNew(1, values)
SheetSetRowLabels({"AA", "BB", "CC", "DD", "EE"})
SheetControl(_Show)
```



	values
AA	0.8376
BB	0.7265
CC	0.4849
DD	0.2054
EE	0.7437
6	0.4685
7	0.4580
8	0.9492
9	0.7444
10	0.1083
11	0.5990
12	0.3852
13	0.7350
14	0.6090

Changing Column Headers

Use the `SheetSetColLabels` command to replace column headers with a set of specified labels as follows:

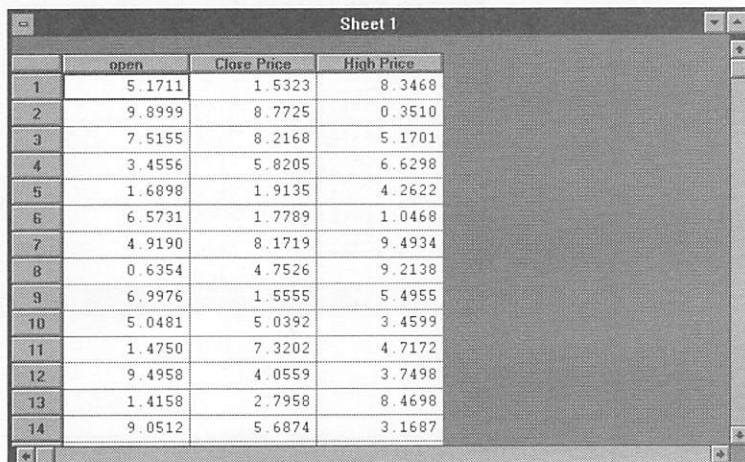
```
SheetSetColLabels [(VarName, Labels)]
```

VarName is the variable name of the column that corresponds to the first label in *Labels*; successive columns will be replaced with the remaining elements of *Labels*. Repeated cells with the same *VarName* will replace old labels.

Note: If `SheetSetColLabels` is called without any parameters, the labels are restored to their original names.

The example that follows illustrates the replacement of column headers with specified labels. The sheet's column headers are first populated with the names of three array variables (*open*, *close*, and *high*) that each contain random numbers. The `SheetSetColLabels` command is then used to change two of the default headers—starting with *close*—to “Close Price” and “High Price”:

```
open = Rnd(20) * 10
close = Rnd(20) * 10
high = Rnd(20) * 10
SheetNew(1, open, close, high)
SheetSetColLabels(close; {"Close Price", \
    "High Price"})
SheetControl(_Show)
```



	open	Close Price	High Price
1	5.1711	1.5323	8.3468
2	9.8999	8.7725	0.3510
3	7.5155	8.2168	5.1701
4	3.4556	5.8205	6.6298
5	1.6898	1.9135	4.2622
6	6.5731	1.7789	1.0468
7	4.9190	8.1719	9.4934
8	0.6354	4.7526	9.2138
9	6.9976	1.5555	5.4955
10	5.0481	5.0392	3.4599
11	1.4750	7.3202	4.7172
12	9.4958	4.0559	3.7498
13	1.4158	2.7958	8.4698
14	9.0512	5.6874	3.1687

Hiding Row and Column Headers

You can hide or show spreadsheet row and column headers with the `SheetControlLabels` command as follows:

```
SheetControlLabels(Action; Which)
```

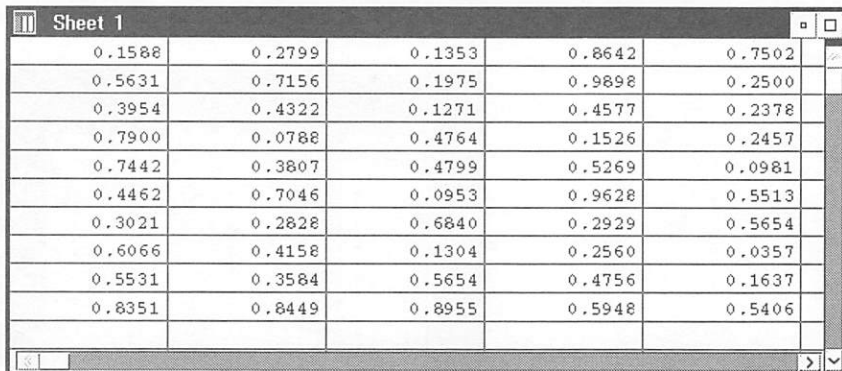
To hide the headers, specify *Action* as `_Hide`. To reveal hidden headers, set *Action* to `_Show`.

Which can be one or a sum of the following constants:

Constant	Hides or Shows
<code>_Row</code>	Row headings
<code>_Col1</code>	First-level column headings
<code>_Col2</code>	Second-level column headings
<code>_All</code>	All headings

The following code produces a sheet with hidden labels:

```
a[1:10, 1:10] = Rnd(100)
SheetNew(1, a)
SheetControlLabels(_Hide; _All)
SheetControl(_Show)
```



0.1588	0.2799	0.1353	0.8642	0.7502
0.5631	0.7156	0.1975	0.9898	0.2500
0.3954	0.4322	0.1271	0.4577	0.2378
0.7900	0.0788	0.4764	0.1526	0.2457
0.7442	0.3807	0.4799	0.5269	0.0981
0.4462	0.7046	0.0953	0.9628	0.5513
0.3021	0.2828	0.6840	0.2929	0.5654
0.6066	0.4158	0.1304	0.2560	0.0357
0.5531	0.3584	0.5654	0.4756	0.1637
0.8351	0.8449	0.8955	0.5948	0.5406

Updating the Contents of a Spreadsheet

Once you have created a sheet, you can update its contents at any time. Using the `SheetUpdate` command, you can add new columns to the sheet by adding more variables, update the existing data if their associated variables have been updated (this is analogous to choosing `Recalculate` in a conventional spreadsheet), and delete cleared variables.

Adding New Variables

After a spreadsheet is created, you can add variables to it with the `SheetUpdate` command. The general form of this command is as follows (note that it updates the current sheet):

```
SheetUpdate [(Data1 [..., DataN]  
            [; Width1 [..., WidthN]])]
```

Data1 through *DataN* are the variables to be added. If any of the specified variables already exist in the current sheet, the cells for those variables are updated. Otherwise, each new variable is added to the end of the spreadsheet (from left to right, in the order in which they were specified).

Note: If no variables are specified, all the columns in the sheet are updated, and any cleared variables are removed.

Width1 through *WidthN* specify the width—in characters—of the *Data1* through *DataN* columns (the widths should be listed in the same order as the variable names to which they correspond). If omitted, the default width of 12 is used for new columns. (*Width* values less than 1 cause a column to return to the default width of 12 characters.)

Recalculating the Data

If a program changes the value of the variables displayed in a sheet, the contents of the sheet *will not* be updated automatically—you must call `SheetUpdate` to refresh the sheet.

Individual sheet cells are always updated whenever they are uncovered or scrolled into view. If the values have been changed without a corresponding `SheetUpdate`, this can cause the data in the sheet to be inconsistent. `SheetUpdate` should be used whenever possible to force the changes to be shown and prevent this inconsistency.

Note: By clicking in a cell and typing a new value, the user can change the data values in a spreadsheet at any time. Remember that this also modifies the actual variable in memory represented by that column. To prevent the user from changing the values of variables, protect them with `SheetControl(_ReadOnly; 1)`.

Removing Cleared Data

If a variable displayed in a sheet is cleared, the data in the column will become invalid. Clicking in a cell will cause CA-Realizer to beep, and clicking on the column heading button will cause the column data to be erased.

If a variable is cleared, use `SheetUpdate` to refresh the sheet. This causes the entire column to be removed.

Example

The following program creates a sheet with information about parts, and then updates it:

```
Parts = {"Nut", "Bolt", "Screw", "Washer"}
Prices = {0.04, 0.06, 0.11, 0.02}
SheetID = SheetQUnique
SheetNew(SheetID, Parts, Prices)
SheetControl(_Show)

'Update the price of screws
Prices[3] = 0.12
SheetUpdate(Prices)

'Add quantity to the sheet. Display it as a whole
'number, and make the column 8 characters wide
Quantity = {1000, 1500, 850, 2200}
SetFormat(Quantity, "P(0)")
SheetUpdate(Quantity; 8)
```

	Parts	Prices	Quantity
1	Nut	0.0400	1000
2	Bolt	0.0600	1500
3	Screw	0.1200	850
4	Washer	0.0200	2200
5			
6			
7			
8			
9			
10			

Querying the Spreadsheet Selection

Often, you will want to manipulate specific areas of the sheet at the same time. The `SheetQInfo` command permits you to select any rectangular area of a sheet, including rows, columns, or all cells, in order to perform some action on them.

At any time, the user can select one or more cells by dragging the mouse over regions of the spreadsheet, or by pressing or dragging on row and column buttons. The program can query this selection to act upon the selected data.

The general form of `SheetQInfo` is shown below:

```
Info = SheetQInfo(Action [, ColNum])
```

Action can be one of the following constants:

Action	Returns
<code>_Selection</code>	A four-element array representing the currently highlighted area of the sheet as <i>{Left, Top, Right, and Bottom}</i> , specified in row and column numbers.
<code>_Range</code>	The range of rows for column <i>ColNum</i> as a two-element array with the starting and ending row values. If the sheet contains a two-dimensional array and <i>ColNum</i> is not specified, the value returned is a four-element array containing <i>{StartCol, StartRow, EndCol, EndRow}</i> .

The next example places a sheet in a form with a Sum button. When the button is pressed, a message box is shown with the sum of the values in the selected region.

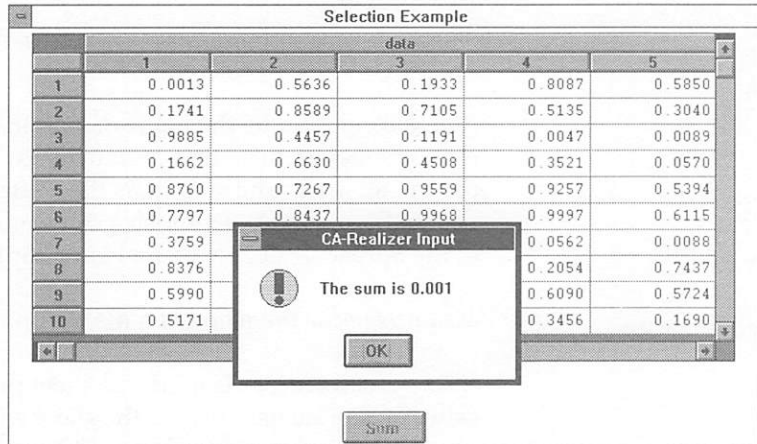
```

' Manual\PG_12\QSELECT.RLZ

PROC formproc(params)
  FormSelect(params[_FormNum])
  SELECT CASE params[_Invoke]
    CASE _Click
      SELECT CASE params[_ItemNum]
        CASE 20
          sel = SheetQInfo(_Selection)
          n = sum(data[sel[2]:sel[4], \
            sel[1]:sel[3]])
          INPUT Sprintf("The sum is P(-3)", \
            n);
        END SELECT
      END SELECT
    END PROC

data[1:10, 1:10] = Rnd(100)
FormNew("Selection Example", _Title)
FormControl(_Size; _Center, _Center, 80 pct, \
  80 pct)
FormSetObject(10, _Sheet, "", _Center, _Top, \
  95 pct, 80 pct)
FormSetObject(20, _Button, "Sum", _Center, \
  _Bottom)
FormSetProc(formproc)
SheetUpdate(data)
SheetControl(_RowRange; 1, 10)
SheetControl(_ColRange; 1, 10)
FormControl(_Show)

```



Setting a Spreadsheet Procedure

When the user tries to close the sheet or double-clicks inside a cell, the sheet's procedure is invoked. You can use a sheet procedure to display more information about a specific cell (or its variable) every time it is double-clicked. You can also design a sheet to invoke the sheet procedure every time the user updates a value in a cell.

Sheet procedures are very powerful and you use them to customize your applications.

Tip: See the "Overview of the CA-Realizer Programmable Application Tools" chapter for information about tool procedures.

Attaching a Spreadsheet Procedure

Use `SheetSetProc` to associate a procedure or program module with the current sheet:

```
SheetSetProc([ProcName | ModuleName]  
             [; [_Size] [, _Change]])
```

ProcName should be the name of a defined procedure that takes two parameters. The first parameter is the tool procedure parameter array; the second, is the actual column of data, passed by reference, in which a cell was clicked (see Processing in the Spreadsheet Procedure below for more information).

ModuleName is the name of a file to run.

If the `_Size` modifier is used, the sheet procedure will also be called when the user resizes the sheet window. If `_Change` is specified, the sheet procedure will be called every time the user modifies a value in a cell.

Note: `SheetSetProc` without any parameters disassociates the current sheet procedure from the sheet.

Processing in the Spreadsheet Procedure

The basic structure of a sheet procedure is as follows:

```
PROC MySheetProc(params, columnVar)
  SheetSelect(params[_ItemNum], \
    params[_FormNum])

  SELECT CASE params[_Invoke]
    CASE _Close
      'Possibly do something when the sheet
      'is closing. To prevent the close, do:
      'params[_UseRealizer] = 0

    CASE _Click
      'Process the mouse double-click here.
      'params[_XPos] is the number of the
      'column in the sheet where the mouse was
      'clicked.
      'params[_YPos] is the row number.

    CASE _Change
      'If _Change was specified with
      'SheetSetProc, this message will come in
  END SELECT
END PROC
```

The second parameter to the sheet procedure is the actual column of data, passed by reference. This means that you can change the value in the cell by modifying the *params[_YPos]* element of the *columnVar* parameter. You can then use *SheetUpdate(columnVar)* to force this change to display immediately.

The value of *params[_XPos]* indicates the column number of the selected cell. Because you can add and remove variables from the sheet, it is not always possible to assume which variable is in which column. Since the second parameter, *columnVar*, is passed by reference, you can get the actual name of the variable with *QVar(column, _Name)*.

As noted in the example, you can prevent the sheet from being closed by setting the *_UseRealizer* element of the sheet procedure parameter array to 0.

A Sample Spreadsheet Procedure

As a quick example of the kind of processing possible with CA-Realizer sheets, here is a program that displays two columns of data—amounts and percents. The amounts column is a set of real numbers. Each element of the percent column represents the value of its row as a percent of the sum of all values in the amounts column. For example, if the amounts column contains {200, 300, 800, 700}, the percents column contains {10, 15, 40, 35}.

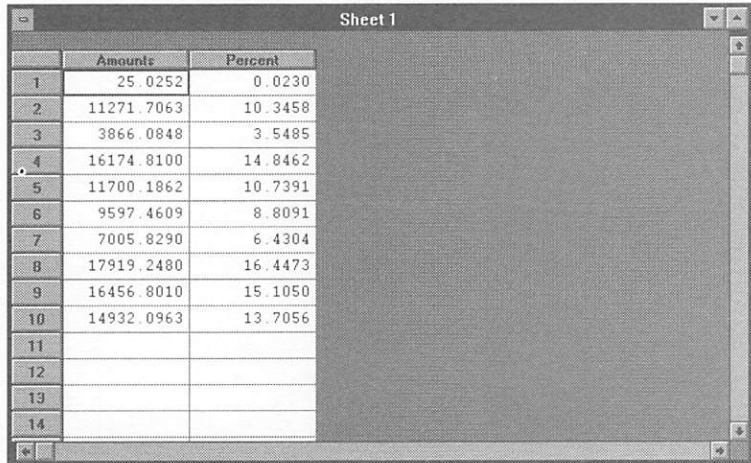
As the user changes the values in the cells, the sheet procedure is called, and a recomputation is done. If the cell changed is an amount cell, the percent column is recomputed based on the current entries in the amounts column. If a percent cell is clicked on, that percent value is held fixed while all the other percents and amounts around it are recomputed.

In order for the sheet procedure to be called on every cell change, the `_Change` modifier has to be specified with the `SheetSetProc` command.

```
' Manual\PG_12\SHEETPRC.RLZ
PROC MySheetProc(params, columnVar)
  LOCAL Row, CellVal

  SheetSelect(params[_ItemNum], \
    params[_FormNum])
  SELECT CASE params[_Invoke]
  CASE _Change
    SELECT CASE params[_XPos]
    CASE 1
      Percent = StatPercent(Amounts) * 100
      SheetUpdate(Percent)
    CASE 2
      Row = params[_YPos]
      CellVal = Percent[Row]
      IF CellVal>=0 AND CellVal<=100 THEN
        Percent[Row] = 0
        Percent = StatPercent(Percent) * \
          (100 - CellVal)
        Percent[Row] = CellVal
        Amounts = Sum(Amounts) * Percent/100
        SheetUpdate(Amounts, Percent)
      END IF
    END SELECT
  END SELECT
END PROC
```

```
Amounts = Rnd(10) * 20000  
Percent = StatPercent(Amounts) * 100  
SheetNew(SheetQUnique, Amounts, Percent)  
SheetControl(_Show)  
SheetSetProc(MySheetProc; _Change)
```



	Amounts	Percent
1	25.0252	0.0230
2	11271.7063	10.3458
3	3866.0848	3.5485
4	16174.8100	14.8462
5	11700.1862	10.7391
6	9597.4609	8.8091
7	7005.8290	6.4304
8	17919.2480	16.4473
9	16456.8010	15.1050
10	14932.0963	13.7056
11		
12		
13		
14		

Chapter 13

Boards

In This Chapter	13-1
Displaying Data in a Board	13-2
Creating a Board	13-3
Specifying a Board Template	13-5
Listing Member Names	13-6
Changing the Title of a Column	13-7
Indicating the Column Format	13-8
Specifying the Layout	13-9
Updating the Contents of a Board	13-10
Adding More Data	13-10
Recalculating the Data	13-12
Removing Cleared Data	13-12
Setting a Board Procedure	13-13
Attaching a Board Procedure	13-13
Processing in the Board Procedure	13-14
A Sample Board Procedure	13-15

Chapter 13

Boards

In This Chapter

CA-Realizer's boardwatch tool (often referred to as a *board*) provides a convenient way to display data in a tabular form. It differs from a spreadsheet in several ways, the most important being that you can only change data in a board via a program. You usually use boards to show data—such as stock information—that is continually changing or updating from an external source.

In this chapter, you will learn how to use the various commands and functions that enable you to control and program boards.

Note: The commands you use to create and manipulate a boardwatch—such as BoardNew, BoardQUnique, BoardControl, and BoardSelect—work like their other tool counterparts. See the “Overview of the CA-Realizer Programmable Application Tools” chapter for details about the standard tool commands.

The following table briefly describes the commands and functions that can be used to create and control boards:

Command	Description
BoardControl	Allows you to control the attributes of the current board.
BoardNew	Creates a new board.
BoardQ	Returns information about a board.

Continued

Continued

Command	Description
BoardQUnique	Returns a unique board number.
BoardSelect	Selects a board to be the current board.
BoardSetProc	Assigns a procedure to the current board.
BoardUpdate	Adds and updates families in a board.

Many of these commands and functions are discussed in this chapter. For more detailed information, refer to the *CA-Realizer Language Reference Guide*.

Displaying Data in a Board

A board displays a set of families, each of which is expected to have the same members. For example, a board can contain families for a set of companies, such as *Acme*, *Smythe Corp.*, *DataTech*, and *CoolCola*. Each of these families contains members representing stock information for that company: *High*, *Low*, *Close*, and *Volume*:

Acme.High = 99.75	Smythe.High = 46.75
Acme.Low = 70.375	Smythe.Low = 34.5
Acme.Close = 75.125	Smythe.Close = 39.875
Acme.Volume = 20000	Smythe.Volume = 15000
DataTech.High = 21.5	CoolCola.High = 73.5
DataTech.Low = 19.875	CoolCola.Low = 69.0
DataTech.Close = 20.125	CoolCola.Close = 73.5
DataTech.Volume = 40000	CoolCola.Volume = 120000

By default, each row of the board represents a single family and each column represents a family member. Below you will see a board containing this information:

Row Based Board				
	High	Low	Close	Volume
Acme	99 3/4	70 3/8	75 1/8	20000
Smythe	46 3/4	34 1/2	39 7/8	15000
Tech	21 1/2	19 7/8	20 1/8	40000
CoolCola	73 1/2	69	73 1/2	120000

You can transpose the rows and columns of a board so that each column represents a family and each row represents a family member:

	Column Based Board			
	Acme	Smythe	Tech	CoolCola
High	99 3/4	46 3/4	21 1/2	73 1/2
Low	70 3/8	34 1/2	19 7/8	69
Close	75 1/8	39 7/8	20 1/8	73 1/2
Volume	20000	15000	40000	120000

Creating a Board

CA-Realizer allows you to create and manipulate boards with commands similar to sheet commands. You can define a new board using `BoardNew`, update it with `BoardUpdate`, and display it using `BoardControl(_Show)`.

Although you use board commands like sheet commands, boards operate differently:

- Boards only work with data organized into families. Depending on the layout of the board, you can list the members of these families in either a row or a column.
- Data displayed in a board can *only* be changed programmatically (that is, the user cannot edit the data displayed in the cells of a board).
- You cannot adjust the various board properties, such as column widths.
- If you add families to a board, the size of the board does not automatically change.

You commonly use Boards to display *live data*, which is data that continuously changes via a CA-Realizer program or an external source (for example, stock information is a type of live data). Using boards, you can build a CA-Realizer program that displays current stock prices and averages from a stock data service, or you can show a sporting event's scoreboard that can be updated for each player as the game progresses.

The general form of the BoardNew command is as follows:

```
[BoardID = ]BoardNew(BoardNumber, Template [, Data1
    [... , DataN]][; Title [, Style]])
```

Like any CA-Realizer tool, each board must have a unique ID. You can use the BoardNew or BoardQUnique command to generate one, or you can specify a particular ID when issuing the BoardNew command.

Template specifies the template to be used for the board, and *Data1* through *DataN* are a list of families that should be placed in the board.

Use *Title* and *Style* to override the board's default attributes. For example, the board's title is initially set to "Board *BoardNum*" (for example, Board 10). Specifying *Title* allows you to set it to a different text string.

Additionally, the default window style of a board is *_Title + _Size + _Minimize + _Close*. Using the *Style* parameter, you can add additional styles or remove some or all default ones. (Available *Style* constants are listed in the *Language Reference Guide*.)

The following information defines three families—*Acme*, *Smythe*, and *Tech*—each representing data for a different company. The members of these families hold current stock information:

```
Acme.High = 99.75
Acme.Low = 70.375
Acme.Close = 75.125
Acme.Volume = 20000

Smythe.High = 46.75
Smythe.Low = 34.5
Smythe.Close = 39.875
Smythe.Volume = 15000

Tech.High = 21.5
Tech.Low = 19.875
Tech.Close = 20.125
Tech.Volume = 40000
```

The examples in this chapter demonstrate how to create and control boards using these three families. In addition, they also use the *MyPlate* template.

Specifying a Board Template

When creating a board, you must also specify a *template*. A template is a special type of family that maintains data related to the board's members, as well as the board's format, layout, and contents.

The members of a template family are listed below. Only the first member is mandatory—the rest are optional.

Member	Description
<i>Template.Member</i>	String array containing a list of the family members that should be displayed in the board fields. If the member name is "", the family's name is used.
<i>Template.Title</i>	String array that contains a list of names to use as the titles for the fields. By default, the member names are used as the titles.
<i>Template.Format</i>	String array that contains a list of format strings to use when displaying the data in each field. By default, the default format of the variable is used.
<i>Template.Layout</i>	Optional scalar value that controls the layout of the board.

Each template member is discussed in more detail in the rest of this section.

Note: When a board is created, it is invisible. Use `BoardControl` to make it visible.

Listing Member Names

The *Template.Member* array is the only mandatory member of a board's template. The strings in this array are the names of family members that should be displayed in the fields of the board.

CA-Realizer searches for these members to see if they belong to the families that the board will display (recall that the families to be displayed in a board are specified in the BoardNew command as *Data1* through *DataN*).

When CA-Realizer finds a match, the value of that member is displayed in the board. Otherwise, "UNDEFINED" appears in the board cell.

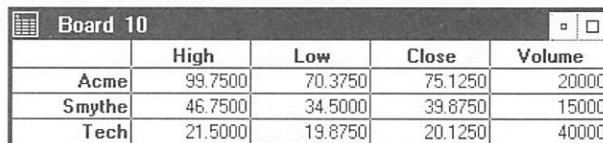
For example, the following creates a *Template.Member* array for the *Acme*, *Smythe*, and *Tech* families (defined earlier in this chapter):

```
MyPlate.Member = {"", "High", "Low", "Close", \\  
                  "Volume"}
```

When the first element of the *Template.Member* array is an empty string (""), the family name becomes a header for the row or column holding the family data. It also sets the *High*, *Low*, *Close*, and *Volume* members as board items.

Now construct a board with this simple template:

```
BoardNew(10, MyPlate)  
BoardUpdate(Acme, Smythe, Tech)  
BoardControl(_Show)
```



	High	Low	Close	Volume
Acme	99.7500	70.3750	75.1250	20000
Smythe	46.7500	34.5000	39.8750	15000
Tech	21.5000	19.8750	20.1250	40000

The new board displays the families in rows. Each row holds stock prices from a different company, and displays the name of a family. The column headers display the name of each family member.

This board uses the default formats for a board. By default, a board displays families by row, and has heading labels for both the families and the family members. Override these defaults by specifying a different layout in the board template.

Tip: The `BoardUpdate` command, which operates like the `SheetUpdate` command, places new families in a board and updates the values of old ones. When data in a board family changes, use the `BoardUpdate` command to display the new information.

Changing the Title of a Column

CA-Realizer automatically displays member names at the head of a board column. You can, however, specify other labels in place of these column headings with the `Template.Title` array. The elements of the `Template.Title` array must correspond directly to the elements of the `Template.Member` array.

For instance, close the current board (if it is still displayed) and modify the program by adding the following statement *after* the `MyPlate.Member` statement but *before* the `BoardNew` statement:

```
MyPlate.Title = {"Company", "High Price", "\\  
"Low Price", "Close Price", "Volume"}
```

Board 10				
Company	High Price	Low Price	Close Price	Volume
Acme	99.7500	70.3750	75.1250	20000
Smythe	46.7500	34.5000	39.8750	15000
Tech	21.5000	19.8750	20.1250	40000

As you can see, the column headings are now more descriptive.

Indicating the Column Format

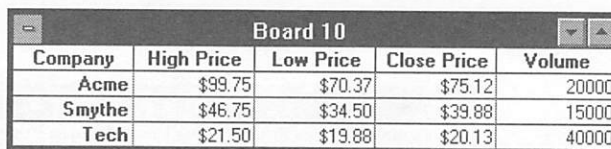
How the data in a board is displayed depends on the format defined for the family members that are associated with that board's data. Each member is of a specific data type; in turn, each data type has a default format string. (A *format string* describes how a value looks when it is displayed.) When a board displays values, they appear in their default format.

Note: Default format strings (as well as all available format strings) are listed in the PRINT command in the *CA-Realizer Language Reference Guide*.

You can, however, format the contents of a board using the *Template.Format* array. This array can contain a set of format strings to be associated with each family member. The next time the member is displayed in a board, it will appear in its new format.

For example, close the current board (if it is still displayed) and modify the program by adding the following statement *after* the *MyPlate.Title* statement but *before* the *BoardNew* statement:

```
MyPlate.Format = { "", "$$##.##", "$$##.##", "\\  
                "$$##.##", "P(0) " }
```



Company	High Price	Low Price	Close Price	Volume
Acme	\$99.75	\$70.37	\$75.12	20000
Smythe	\$46.75	\$34.50	\$39.88	15000
Tech	\$21.50	\$19.88	\$20.13	40000

The *High*, *Low*, and *Close* columns of the board are now written as dollar amounts.

Note: The first string of the *Template.Format* array is left empty because CA-Realizer cannot change the format of the family names.

If the format string begins with a question mark (for example, ?P(-2)), the negative numbers in that column will be displayed in red. This can be useful when the sign of the value in the column is important, such as for a profit-loss value.

Specifying the Layout

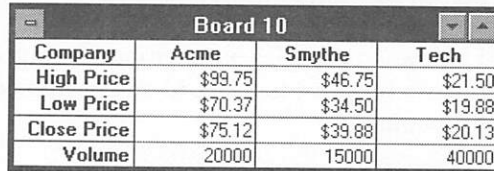
So far, you have been using the default listing of families by row. You can, however, transpose this layout so that families are grouped by columns and members are listed in rows.

Use the *Template.Layout* member to control the layout of a board. It can be the sum of one or more of the following constants:

Constant	Description
<code>_FamilyRowwise</code>	Causes each row to represent a family, and each column to represent a family member.
<code>_FamilyColwise</code>	Causes each column to represent a family, and each row to represent a family member.
<code>_NoFieldHeaders</code>	Prevents the display of heading labels for each family member.
<code>_NoFamilyHeaders</code>	Prevents the display of heading labels for each family.
<code>_Default</code>	Uses the default settings (that is, <code>_FamilyRowwise</code> and all heading labels visible).

For example, close the current board (if it is still displayed) and modify the program by adding the following statement *after* the `MyPlate.Format` statement but *before* the `BoardNew` statement:

```
MyPlate.Layout = _FamilyColwise
```



Company	Acme	Smythe	Tech
High Price	\$99.75	\$46.75	\$21.50
Low Price	\$70.37	\$34.50	\$19.88
Close Price	\$75.12	\$39.88	\$20.13
Volume	20000	15000	40000

Notice how the families displayed in the board are transposed. Additionally, clicking in a cell from this new board selects an entire column.

Updating the Contents of a Board

Once you have created a board, you can update its contents at any time. Using the `BoardUpdate` command, you can add new columns to the board by adding more families, updating (or recalculating) the existing data, and deleting cleared data.

Adding More Data

You can add families to a board with the `BoardUpdate` command. The general form of this command is as follows (note that it updates the current board):

```
BoardUpdate [(Data1 [..., DataN ])]
```

Data1 through *DataN* are the names of families to be added. If a family member is specified and already exists in the current board, its cell is updated. Otherwise, each new family is added to the end of the board (from left to right, in the order in which they were specified).

Note: You cannot add individual members to the board in this way. Additionally, `BoardUpdate` without any parameters will update all the families in a board.

When you add families to a board, the size of the board *does not* automatically change. You can, however, recompute the width and height of a board window based on the families it contains by using the `BoardControl(_Size)` command with the `Width` and `height` parameters set to `_Default`.

Type in and execute the following lines to add a fourth company to the stock board:

```
Tronics.High = 55.75
Tronics.Low = 52.375
Tronics.Close = 54.125
Tronics.Volume = 80000
BoardUpdate(Tronics)
```

Notice that the window size has not changed. To see the new family, drag the window border to make the board window larger. Resize the board window now. CA-Realizer lists the members of the *Tronics* family in the last row.

As you add families to a board, the size of the board does not automatically change. You can, however, recompute the width and height of a board window based on the families it contains by using the `BoardControl(_Size)` command with the parameters set to `_Default`.

```
BoardControl(_Size; _Default, _Default, \\  
_Default, _Default)
```

Recalculating the Data

If a program changes the values displayed in a board without a corresponding BoardUpdate, individual board cells will be updated if the user exposes a cell by uncovering it from underneath another window. Use BoardUpdate whenever a program changes a value to prevent this inconsistency.

Removing Cleared Data

If you clear a family or family member that is displayed in a board with the CLEAR command, its data becomes invalid, and the corresponding board cells will be blank if redisplayed. Any subsequent BoardUpdate command removes this family.

Note, however, that CA-Realizer will not automatically adjust the size of the board.

Setting a Board Procedure

When the user tries to close the board or double-clicks inside a family row or column, the board's procedure is invoked. You can use a board procedure to display more information about a specific family.

Unlike sheets, CA-Realizer highlights an entire row or column representing a family by default when the board is selected. While CA-Realizer passes information about this entire family to the board, it will not be able to determine in which specific cell the user clicked.

Board procedures are very powerful and you use them to customize your applications.

Tip: See the "Overview of the CA-Realizer Programmable Application Tools" chapter for information about tool procedures.

Attaching a Board Procedure

Use `BoardSetProc` to associate a procedure or program module with the current board:

```
BoardSetProc [(ProcName | ModuleName)][;_Size]
```

ProcName should be the name of a defined procedure that takes two parameters. The first parameter is the tool procedure parameter array; the second is the actual family, passed by reference, in which the user clicked (see Processing in the Board Procedure below for more information).

ModuleName is the name of a module file to run.

If the `_Size` modifier is used, the board procedure will also be called when the user resizes the board window.

Note: `BoardSetProc` without any parameters disassociates the current board procedure from the board.

Processing in the Board Procedure

The basic structure of a board procedure is as follows:

```
PROC MyBoardProc(params, familyVar)
  BoardSelect(params[_ItemNum], params[_FormNum])
  SELECT CASE params[_Invoke]
    CASE _Close
      'Possibly do something when the board is
      'closing. To prevent the close, do:
      'params[_UseRealizer] = 0
    CASE _Click
      'Process the mouse double-click here.
      'The family parameter will be the family
      'that was clicked in, passed by reference.
  END SELECT
END PROC
```

The second parameter to the board procedure is the actual family, passed by reference. This means that you can change the values in the family by modifying the members of the *familyVar* parameter. You can then use `BoardUpdate(familyVar)` to force this change to be displayed immediately.

Since the second parameter, *familyVar*, is passed by reference, you obtain the actual name of the family with `QVar(Family, _Name)`.

As noted in the example, you can prevent the board from being closed, by setting the `_UseRealizer` element of the board procedure parameter array to 0.

A Sample Board Procedure

Consider a board containing stock information. Each family in the board is a stock name with four members: *High*, *Low*, *Close*, and *Volume*. The stock information comes from a live feed providing the latest prices for a given stock.

Since the data changes rapidly, it would be inefficient to update the values continually. While a program might want to update periodically, a board can also be designed to update a stock column when it is double-clicked, as shown in the following example:

```
PROC UploadNewData(familyVar)
    LOCAL StockName, NewLow, NewHigh, NewClose, \
           NewVolume
    StockName = QVar(familyVar, _Name)
    'Upload the new information from an outside
    'source, such as a live stock communications
    'feed. The function call might look like
    'this:
    ' StockRead(StockName, NewHigh, NewLow,
    '           NewClose, NewVolume)

    familyVar.High = NewHigh
    familyVar.Low = NewLow
    familyVar.Close = NewClose
    familyVar.Volume = NewVolume
END PROC

PROC MyBoardProc(params, familyVar)
    BoardSelect(params[_ItemNum], params[_FormNum])
    SELECT CASE params[_Invoke]
        CASE _Click
            UploadNewData(familyVar)
            BoardUpdate(familyVar)
    END SELECT
END PROC
```


Chapter 14

Text Logs

In This Chapter	14-1
Creating a Log	14-2
Displaying Information in a Log	14-3
Controlling the Appearance of a Log	14-5
Clearing a Log	14-5
Setting a Font	14-6
Setting Tab Stops	14-7
Obtaining Information About a Log and Its Contents	14-8
Determining the Number of Characters	14-9
Extracting Text From a Log	14-9
Extracting a Line of Text From a Log	14-9
Manipulating the Information in a Log	14-10
Setting a Log Procedure	14-12
Attaching a Log Procedure	14-12
Processing in the Log Procedure	14-13
A Sample Log Procedure	14-13

Chapter 14

Text Logs

In This Chapter

Text logs are used for displaying text. The PRINT command can send its output to a text log and the user can enter and edit text within a log using the standard editing features such as cutting and pasting. This chapter describes the log tool in greater depth.

Note: The commands you use to create and manipulate a text log—such as LogNew, LogQUnique, LogControl, and LogSelect—work like their other tool counterparts. See the “Overview of the CA-Realizer Programmable Application Tools” chapter for details about the standard tool commands.

The following table briefly describes the commands and functions that can be used to create and control logs:

Command	Description
LogControl	Allows you to control the attributes of the current log.
LogNew	Creates a new log.
LogQ	Returns information about a log.
LogQData	Returns information about the contents of a log.
LogQSize	Returns the number of characters in the current log.
LogQStr	Returns text from the current log.
LogQUnique	Returns a unique log number.

Continued

Continued

Command	Description
LogSelect	Selects a log to be the current log.
LogSetData	Changes the contents of a log.
LogSetProc	Assigns a procedure to the current log.

Many of these commands and functions are discussed in this chapter. For additional information, refer to the *CA-Realizer Language Reference Guide*.

Creating a Log

CA-Realizer has two system logs: the Error Log and the Print Log. CA-Realizer invokes the Error Log whenever an error condition occurs, and sends PRINT command output to the Print Log.

Using LogNew, you can create as many logs as you want. Like any CA-Realizer tool, each log must have a unique ID. You can use the LogNew or LogQUnique command to generate one, or you can specify a particular ID when issuing the LogNew command.

The general form for creating a log is as follows:

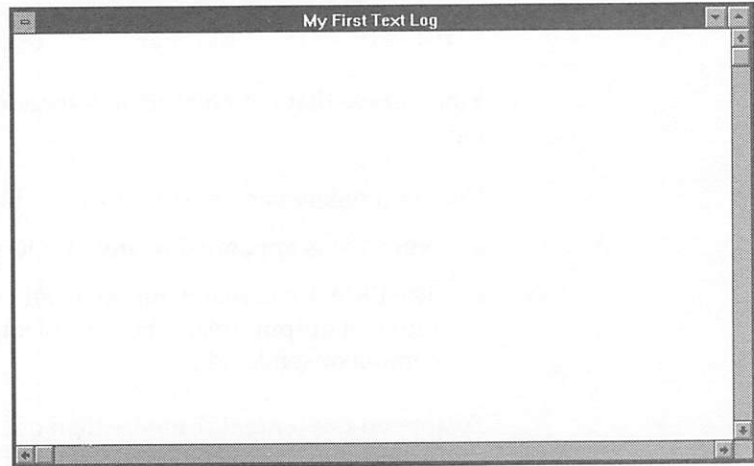
```
[ID = ] LogNew([LogNum][; Title [, Style]])
```

Use *Title* and *Style* to override the log's default attributes. The log's title is initially set to "Log LogNum" (for example, Log 10). Specifying *Title* allows you to set it to a different text string.

Additionally, the default window style of a log is _Title + _Size + _Minimize + _Close. Using the *Style* parameter, you can add additional styles or remove some or all default ones. (Available *Style* constants are listed in the *Language Reference Guide*.)

For example:

```
LogID = LogNew( ; "My First Text Log")
LogControl(_Show)
```



Text Logs

Note: When a log is created, it is invisible. LogControl must be used to make it visible.

Displaying Information in a Log

In addition to directing output from PRINT commands to CA-Realizer's Print Log, you can display data in a text log by specifying the log's ID in the PRINT command.

```
PRINT #LogNumber; Text
```

For example, the following program creates two text logs and displays the specified string in them:

```
MyLog = LogNew
PRINT #MyLog; "I was walking on the moon one day."
LogControl(_Show)

LogNew(10)
PRINT #10; "We hold these truths to be self-evident"
LogControl(_Show)
```

Note: The # character must precede the specified log ID.

Select each of the log windows to see the text that they contain, and then return to the program window and type and run the next two lines:

```
PRINT #MyLog; "In the merry, merry month of May."  
PRINT #10; "That all men are created equal"
```

You will see that both text strings are added to their respective logs.

Note that whenever you print to a text log or the Print Log:

- New text is appended to any previous text in the window.
- The PRINT command automatically inserts carriage returns into text output unless the expression list ends with a comma or semicolon.

To append the contents of more than one PRINT command on a single line, add a semicolon to the end of each PRINT statement. For example:

```
PRINT #10; "And are endowed";  
PRINT #10; " by their creator with";  
PRINT #10; " certain inalienable rights"
```

Once a log has been displayed, the user can change or delete the text within it. In addition, text can be placed in a log by typing directly into a log from the keyboard, or by using the Clipboard to paste data into a log.

Note: Refer to “Controlling Input and Output” in this guide for more information about the PRINT command.

Controlling the Appearance of a Log

As with the other tools, you control the appearance of a log with the log-specific version of the *TOOLControl* command, *LogControl*.

All of the standard *Action* values that are common to all the tools—such as *_Size*, *_Show*, and *_Minimize*—can be used. (See the “Overview of the CA-Realizer Programmable Application Tools” chapter for details.)

In addition, there are a few log-specific *Action* constants, which are listed below:

Constant	Description
<i>_Clear</i>	Clears all of the text from the current log.
<i>_SetFont</i>	Changes the log font to any of the available typefaces and styles in your Windows or OS/2 environment.
<i>_SetTabs</i>	Sets the tab stops in the current log.

Clearing a Log

_Clear

To clear all of the text from the current log, use *LogControl* with the *_Clear* constant.

For instance, to clear Log 10, run the following:

```
LogSelect(10)  
LogControl(_Clear)
```

Keyboard/Mouse

The user can also clear the text of the log with the EDIT CLEAR ALL menu command or the CTRL+SHIFT+DEL key combination.

Setting a Font

When you create a new log, its text is displayed in the same font as a CA-Realizer program window (as set in CA-Realizer .INI file). You can change this font to any of the available typefaces and styles in your Windows or OS/2 environments.

Since there are many typefaces, point sizes, and style combinations that describe a font, CA-Realizer provides a series of commands that define and manipulate fonts. They are described in the “Using Fonts and Pictures in Tools” chapter of this guide.

Once you define a font, you can apply it to text in logs. You can also apply it to other interface elements, such as buttons and list boxes.

First create the font and then use the LogControl command to set that font for use with the current log. Use the `_SetFont` constant and a modifier containing the font ID. For example, the following statement changes the font in the current log to *Font1*:

```
LogControl(_SetFont; Font1)
```

The log text now appears in this font. If the font number you supply is 0, the log font is reset to the program window font:

```
LogControl(_SetFont; 0)
```

Note: You cannot mix different font styles in a single log window.

Setting Tab Stops

The `_SetTabs` constant enables you to set the tab stops in the current log. Use this constant in the `LogControl` command along with a single modifier indicating the distance between tabs (specify this distance using the CA-Realizer position notation).

For example, the `PRINT` command below displays three arrays in Log 10:

```
PRINT #10; Index(10), Rnd(10), Rnd(10)
```

Now set new tab stops and watch the columns in Log 10 adjust:

```
LogControl(_SetTabs; 33 pct)
```

You need to exercise some caution when using proportional fonts and tabs to display tabular data in a log. Each letter in a proportional font has a different width and, as a consequence, you can have columns out of alignment. This occurs because tabs are set relative to the end of the data in the preceding column. To rectify this situation, pad your columns with blanks or use a monospaced font, such as Courier.

Obtaining Information About a Log and Its Contents

You can use the `LogQ` and `LogQData` functions to get information about a log. `LogQ` provides information about the log itself, such as its size and title, while `LogQData` provides information about the contents of the log.

The most common use for `LogQData` is to extract text from a log. The general form for `LogQData` is:

```
ReturnValue = LogQData(Action [, Val])
```

Action specifies the type of information to get—for example, you can retrieve the text of the currently highlighted selection within the current log or all the text in the current log. (All available *Action* constants are listed in the *Language Reference Guide*.)

Val, used only with certain values of *Action*, specifies either the line number or the character number. The value passed to *ReturnValue* is specific for each type of *Action*.

Several examples of how `LogQData` can be used are presented below.

Note that CA-Realizer returns the requested information oriented as positions, characters, and lines.

Position

A *position* is a point between two characters. The first position, to the left of the first character, is position 0.

Characters

Characters are numbered beginning with 1. In most cases, lines will end with two hidden characters—a carriage return and a line feed (also called a CRLF), which are ASCII 13 and ASCII 10, respectively. These characters are counted when numbering characters and positions.

Lines

Lines are numbered beginning with 1.

Determining the Number of Characters

The `LogQData(_Log_StrSize)` function determines the number of characters in a log. To find the length of text in Log 1, enter the following:

```
LogSelect(1)
TextLength = LogQData(_Log_StrSize)
PRINT TextLength
```

Extracting Text From a Log

Extract text from the current log into a string variable called *TheText* using the `LogQData(_Log_Str)` function:

```
TheText = LogQData(_Log_Str)
PRINT TheText
```

Printing *TheText* completes the transfer of text from log 1 to the Print Log.

Extracting a Line of Text From a Log

`LogQData(_Log_LineStr)` extracts a line of text from a log:

```
PartText = LogQData(_Log_LineStr, 10)
PRINT PartText
```

This example extracts line 10 from the log.

Manipulating the Information in a Log

You can use LogQData to return portions of text, all of the text, or information about the text in a log. Similarly, you can use the LogSetData command to change portions of text, all of the text, or certain characteristics of text in a log.

Its general form is:

```
ReturnValue = LogSetData(Action [, Param1  
[, Param2]])
```

ReturnValue is a return value specific for each *Action*, which specifies the type of information to get. *Param1* and *Param2* are used with certain values of *Action* and specify different parameters.

The following table lists valid *Action* values for LogSetData:

Constant	Description
_Log_DeleteLine	Deletes the line of text specified by line <i>Param1</i> .
_Log_DeleteSelection	Deletes the selected text from the log.
_Log_InsertLine	Inserts a line of text specified by string <i>Param2</i> , before line number <i>Param1</i> . If <i>Param1</i> is greater than the number of lines in the log, <i>Param2</i> is appended to the text in the log.
_Log_LimitText	Limits the number of characters that the user can enter in the current log to <i>Param1</i> characters.
_Log_LineStr	Replaces the line specified by <i>Param1</i> with the string specified in <i>Param2</i> . If <i>Param1</i> is greater than the number of lines in the log, string <i>Param2</i> is appended to the end of the text in the log.

Continued

Continued

Constant	Description
<code>_Log_ScrollLine</code>	Scrolls the text vertically by the number of lines specified by <i>Param1</i> . If <i>Param1</i> is positive, it scrolls down. If <i>Param1</i> is negative, it scrolls up.
<code>_Log_ScrollTo</code>	Scrolls the log to make the line specified by <i>Param1</i> the first visible line.
<code>_Log_Selection</code>	Selects all characters that are within a starting point specified by <i>Param1</i> and an ending point specified by <i>Param2</i> . If <i>Param1</i> = 0 and <i>Param2</i> = -1, all the text in the log is selected. If <i>Param1</i> = -1, any current selection is removed.
<code>_Log_SelectLine</code>	Selects the line number specified by <i>Param1</i> .
<code>_Log_SelectStr</code>	Replaces all the selected text in the log with string <i>Param1</i> .
<code>_Log_Str</code>	Replaces all the text in the log with string <i>Param1</i> .

For example:

```
'Create a log
LogNew(10)
PRINT #10; "This text will be added to the log"
PRINT #10; "This is another line of text."
LogControl(_Show)

'Select all characters between 10 and 34
LogSetData(_Log_Selection,{10, 34})

'Replace the selected text with a new string
LogSetData(_Log_SelectStr, "has been replaced")
```

Setting a Log Procedure

By default, CA-Realizer invokes a log procedure *only when* the user tries to close the log window. This allows the program to ask the user, for example, whether he or she wants to save the text in the log before the contents of the log are deleted. You can also invoke a log procedure when the user resizes the log.

Log procedures are very powerful and you use them to customize your applications.

Tip: See the “Overview of the CA-Realizer Programmable Application Tools” chapter for information about tool procedures.

Attaching a Log Procedure

Use `LogSetProc` to associate a procedure or program module with the current log:

```
LogSetProc [(ProcName | ModuleName)][;_Size]
```

ProcName should be the name of a defined procedure that takes one parameter (the tool procedure parameter array).

If the `_Size` modifier is used, the log procedure will also be called when the user resizes the log window.

Note: `LogSetProc` without any parameters disassociates the current log procedure from the log.

Processing in the Log Procedure

The basic structure of a log procedure is as follows:

```
PROC MyLogProc(params)
  LogSelect(params[_ItemNum], params[_FormNum])
  'Possibly do something when the Log is closing.
  'To prevent the close, do:
  '    params[_UseRealizer] = 0
END PROC
```

Since the only way to invoke a log procedure is to close the log, there is no need to use a SELECT statement based on the value of *params[_Invoke]* (unless the *_Size* modifier was specified with *LogSetProc*).

As noted in the example, you can prevent the log from being closed by setting the *_UseRealizer* element of the log procedure parameter array to 0.

A Sample Log Procedure

A standard use of a log procedure is to verify that the user actually wants to close the log or to provide the opportunity to save the text. This prevents the user from accidentally losing data.

The following log procedure is invoked if the user tries to close the log. The user is first asked if the text of the log is to be saved. If so, the user is prompted for a file name and the data is saved.

Then the user is asked if he or she really wants to close the log. If not, the program sets the *params[_UseRealizer]* value to zero to prevent the close.

```
' Manual\PG_14\LOGPROC.RLZ

PROC MyLogProc(params)
  LOCAL FileName, FileNum

  LogSelect(params[_ItemNum], params[_FormNum])
  IF MessageBox("Save text before closing?", \
    "Log Example", _MB_YesNo) = _Yes
    FileName = StdSaveAs("Text.txt")
    IF FileName <> "" THEN
      FileNum = FileQUnique
      FileOpen(FileNum, FileName, _Write)
      FileWrite(FileNum, LogQStr)
      FileClose(FileNum)
    END IF
  END IF

  IF MessageBox("Close the log?", \
    "Log Example", _MB_YesNo) = _No
    params[_UseRealizer] = 0
  END IF
END PROC

LogNew(LogQUnique)
LogControl(_Show)
LogSetProc(MyLogProc)
```

Chapter 15

Forms

In This Chapter	15-1
What Is a Form?	15-1
Form Commands and Functions	15-2
Form Objects	15-3
Buttons	15-3
Option Buttons	15-4
Option Bitmaps	15-4
Check Boxes	15-4
Check Bitmaps	15-5
Group Boxes	15-5
Edit Controls	15-5
Captions	15-6
Bitmaps, Metafiles, and PCX Pictures	15-6
List Boxes	15-7
Frames	15-8
Scroll Bars	15-8
Digital Clocks	15-9
Event Timers	15-9
OLE Objects	15-9
CA-Realizer Tools	15-9
Creating a New Form	15-10
Controlling a Form	15-14
Obtaining Information About a Form	15-17
Setting the Colors in a Form	15-18
Setting the Background Color	15-19
Setting the Color of an Object	15-19

Adding Form Objects	15-20
Using the FormSetObject Command	15-21
Adding Buttons	15-25
Adding Option Buttons	15-25
Adding Bitmap Option Buttons	15-26
Adding Check Boxes	15-27
Adding Bitmap Check Boxes	15-27
Adding Group Boxes	15-28
Adding Edit Controls	15-28
Adding Captions	15-29
Adding Pictures	15-30
Adding List Boxes	15-31
Adding File Names to a List Box	15-32
Adding Variable Names to a List Box	15-32
Adding Family Names to a List Box	15-32
Adding Font Names to a List Box	15-33
Adding Font Sizes to a List Box	15-33
Adding Text to a List Box	15-33
Combining Modifier Groups	15-34
Adding Frames	15-34
Adding Scroll Bars	15-34
Adding Digital Clocks	15-36
Adding Event Timers	15-36
Adding OLE Objects	15-37
Adding CA-Realizer Tools	15-37
Positioning Objects in a Form	15-38
Manipulating and Modifying Form Objects	15-39
Closing an Object	15-41
Changing the State of an Object	15-41
Changing the Color of an Existing Object	15-42
Setting the Focus to an Object	15-42
Adjusting the Front-to-Back and Tab-Stop Ordering of Objects	15-43
Changing the Status of an Option Button or Check Box	15-44
Modifying a List Box	15-44
Modifying a Scroll Bar	15-45
Modifying an Event Timer	15-45

Querying the Object	15-46
Querying the Object Value	15-47
Obtaining the Numeric Value of an Object with FormQNum	15-47
Obtaining the Text Value of an Object with FormQStr	15-48
Creating Toolbars, Status Bars, and Palettes Using Non-MDI Forms	15-50
Non-MDI Forms	15-50
Adjusting the MDI Region	15-51
Creating a Toolbar or Status Bar	15-51
Creating a Palette	15-52
Creating a Pop-up Form	15-53
Form Processing	15-54
Notification Messages	15-55
Button Notifications	15-55
Option Button and Check Box Notifications	15-56
Edit Control Notifications	15-56
Group Box and Caption Notifications	15-56
Bitmap and Picture Notifications	15-57
Frame Notifications	15-57
List Box Notifications	15-57
Event Timer Notifications	15-57
Scroll Bar Notifications	15-58
Graphic Tablet and Animation Notifications	15-58
Chart, Sheet, Board and Log Notifications	15-58
Enter and Esc Notifications	15-58
Modal Forms	15-59
Edit Control Verification	15-59
Modal Pick and PickDrag	15-61
An Example of a Modal Form	15-62
Modeless Forms	15-65
Form Procedures	15-66
The Structure of the Form Procedure	15-66
Modeless Form Messages	15-67
Edit Control Verification	15-69
Modeless Pick and PickDrag	15-69
An Example of a Modeless Form	15-70

Setting a Background Grid	15-72
Adding Accelerator Keys	15-73
Drag and Drop	15-75
Setting Draggable Objects and Drop Receivers	15-76
Processing Drag and Drop Messages	15-77
A Drag and Drop Example	15-77

Chapter 15

Forms

In This Chapter

In CA-Realizer, forms are used to build powerful and intuitive user interfaces for your CA-Realizer applications. A form provides an important unifying element that allows you to better control program flow by tying together the many CA-Realizer application tools, and by integrating standard user interface controls, such as buttons, list boxes, and edit controls.

In this chapter, you will learn how to use forms to enhance the appearance of the user interface and expand the flexibility of your CA-Realizer programs. Information related to creating, modifying, and processing forms, including the available CA-Realizer form commands, functions, form objects, and form types, is presented.

What Is a Form?

A form is a single window that you can use as a container for many other objects. You can use forms to link application elements together when creating large and sophisticated programs. Forms are designed to provide a mix of both functional sophistication and aesthetic appeal.

When you first create a form, it is like a blank slate on which you can place interface elements. These elements can be sized and positioned anywhere on the form. You can combine these elements into functional groups or associate them with each other through visual clues.

Because forms are versatile, there are several types you can design. For some programs, you can implement the entire user interface with a single form, while other programs link several forms together. A form can wait passively for user interaction, like other CA-Realizer application tools, or it can suspend an application until the user responds.

Forms are created and controlled with the CA-Realizer form commands and functions. These commands and functions are presented below.

Form Commands and Functions

The following table lists the available CA-Realizer commands and functions that allow you to create and control forms. Many of these commands and functions are discussed in this chapter. For additional information, refer to the *CA-Realizer Language Reference Guide*.

Command	Description
FormControl	Controls the attributes of the current form.
FormModifyObject	Modifies the state or appearance of a form object.
FormNew	Creates a new form.
FormQ	Returns information about the current form.
FormQNum	Returns the numeric value of a form object.
FormQObject	Returns an array holding the attributes of a form object.

Continued

Continued

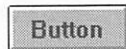
Command	Description
FormQStr	Returns the text value of a form object.
FormQUnique	Returns a unique form number.
FormSelect	Selects a form to be the current form.
FormSetColor	Sets the colors used in the current form.
FormSetObject	Places an object in the current form.
FormSetProc	Assigns a procedure to the current form.
FormWait	Displays the current form, and waits for user interaction.

Form Objects

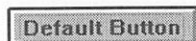
There are a wide variety of Windows and OS/2 controls, called *form objects*, that you can place in a form. These objects are described below.

Buttons

The most basic form object is the *button*. Buttons react when the user chooses them.

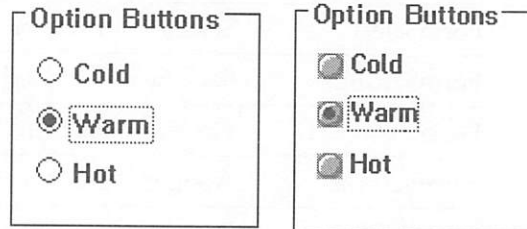


Some buttons have a thicker border around them to indicate they are the default action that occurs when the user presses the ENTER key. These are called *default buttons*.



Option Buttons

Option buttons present a set of choices to the user, only one of which can be selected at a time. When the user chooses a new button, the previously selected button is turned off.



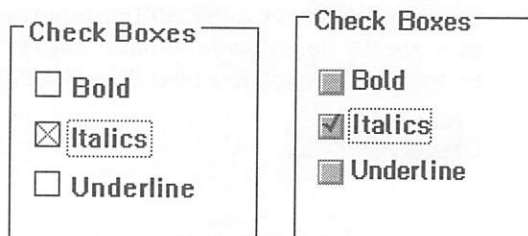
Option Bitmaps

Option bitmaps work the same as option buttons, except that two bitmaps are used to indicate the on and off states. The two states are stored as a single side-by-side image. The left half of the bitmap is extracted and used to represent the on state. The right half is used to represent the off state.



Check Boxes

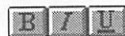
Check boxes present a set of choices to the user, any of which can be on or off at any time. When the user chooses a check box, its state is toggled as indicated by an X or a check mark in the box.



Check Bitmaps

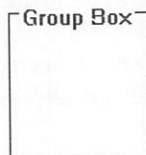
Check bitmaps work the same as check boxes, except that two bitmaps are used to indicate the on and off states. The two states are stored as a single side-by-side image. The left half of the bitmap is extracted and used to represent the on state. The right half is used to represent the off state.

 — This set of check bitmaps is ON.

 — This set of check bitmaps is OFF.

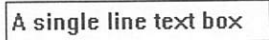
Group Boxes

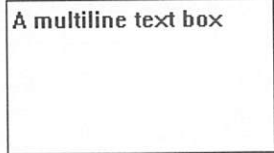
A *group box* is a rectangle used to indicate that a certain set of objects are related. Option buttons and check boxes are often contained in a group box.



Edit Controls

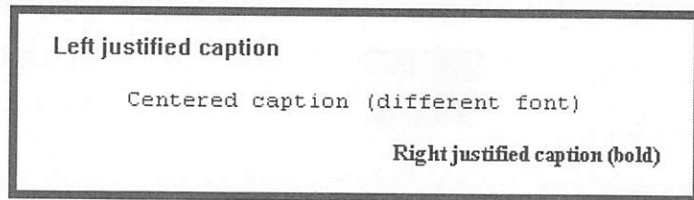
Edit controls can be used to allow the user to enter data from the keyboard. Edit controls are often a single line, but can also be used to enter multiple lines of text. Text within an edit control can be edited using the standard mouse and menu commands.

 A single line text box

 A multiline text box

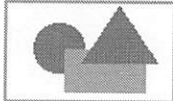
Captions

Captions allow you to place simple text, in any font or style, in a form. There are three types of captions: left-justified, right-justified, and centered. Captions cannot be edited by the user.



Bitmaps, Metafiles, and PCX Pictures

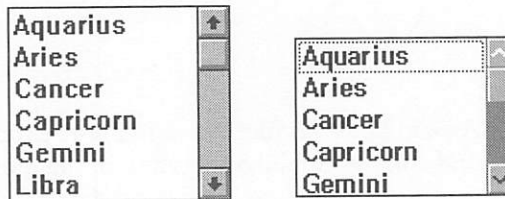
You can place graphical images in the form as *bitmaps*, *metafiles*, or *PCX files*. *Bitmap buttons* can be used to display a simple graphic (for example, a logo) or they can function like other buttons, inverting when they are selected. If the user moves the mouse off a bitmap button before releasing the mouse button, the selection is canceled.



List Boxes

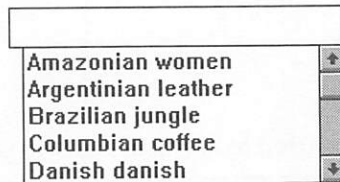
A *list box* allows you to provide a list of choices to the user. The user can scroll through the list of items, and select one. There are four list box types including a regular list box, combo box, drop-down list box, and drop-down combo box.

A *regular list box* simply presents a list of choices for the user to select from.



A *combo box* is a list box with an edit control attached to the top. The user can select one of the choices or type in his or her own. The list box portion scrolls to the closest match to the typed text.

Combo Box



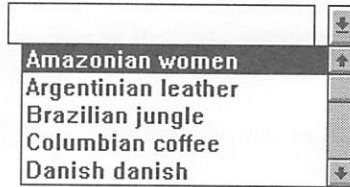
A *drop-down list box* can be used to conserve space in the form. The current selection is displayed at the top, and when the arrow button is pressed, the list box appears and allows the user to make a new selection.

Drop Down List



Finally, a *drop-down combo box* is a drop-down list box with an edit portion and an arrow to open the list box of selections.

Drop Down Combo



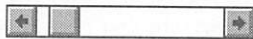
Frames

A sizing *frame* is like the sizing border on a window. The edges and corners can be dragged with the mouse to adjust its size, position, and shape. You can use frames to allow the user to change the size of an object by temporarily representing it as a frame.



Scroll Bars

Scroll bars are used to allow the user to select from a range of values. The user can move the thumb control left and right by clicking on the arrows, or a page at a time by clicking in the scroll region.



Digital Clocks

Digital clocks display the current system time in any valid date/time format.

12:21:32

Note: The default system time is displayed in 24 hour format.

Event Timers

Event timers are set to a specific number of seconds, and then countdown towards zero. When the counter reaches zero, the form is notified and an action can occur.

01:13

OLE Objects

Objects representing elements of other applications can be placed in a form and controlled by another application through *Object Linking and Embedding* (OLE). When an OLE object is double-clicked, the server application that created the object is invoked, allowing the user to change the data in the object.

CA-Realizer Tools

You can also place charts, sheets, boards, logs, graphic tablets, and animation windows in a form. This allows them to be combined with the form controls to make data entry windows and display windows.

Creating a New Form

To create a new form, use the `FormNew` command:

```
[FormNum = ] FormNew([FormNum] [, Title [, Style]])
```

FormNum is the form number used to refer to the form, and should be between 1 and 32767. If a form with that number already exists, the old one is destroyed before the new one is created. If *FormNum* is 0 or unspecified, a unique form number is selected automatically.

Title is the title of the form and *Style* the style of the form. The default style for forms is `_Frame + _Close + _Minimize`.

`FormNew` takes all of the standard *TOOLNew Style* parameters such as `_Title`, `_Size`, and `_Close`. (For more information, see “Overview of the CA-Realizer Programmable Application Tools” in this guide). In addition, `FormNew` features a number of *Style* parameters that are unique to forms.

The following table lists these additional *Style* parameters. They provide a considerable degree of power and flexibility, and include the definition of specialized form window types.

<i>Style</i>	<i>Description</i>
<code>_Borderless</code>	Creates the window without a border (cannot be used in conjunction with <code>_Title</code> or <code>_Size</code>).
<code>_ContextEnter</code>	Controls the behavior of ENTER in the form. By default, ENTER is the equivalent of ID 1 in the form. If ENTER is needed in logs or multiline edit controls, CTRL+ENTER must be used. ContextEnter overrides this default in the form, allowing logs and edit controls to receive the ENTER key. If ContextEnter is used, ENTER no longer notifies the form as ID 1.

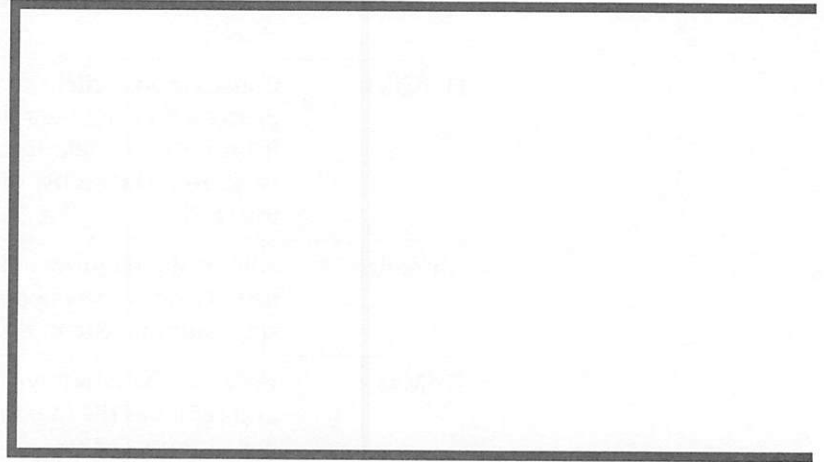
Continued

Continued

Style	Description
<code>_Frame</code>	Places a frame around the form window (cannot be used in conjunction with <code>_Title</code> or <code>_Size</code>).
<code>_HotClick</code>	Causes mouse clicks in the form to be processed even when the form is not the frontmost. By default, a mouse click in a background window only brings that window forward.
<code>_Minimize</code>	Allows the window to be minimized from the form window's system menu. If <code>_Title</code> is also set, a minimize icon is placed in the title bar.
<code>_NoMax</code>	Prevents the window from being maximized and removes the maximize icon from the title bar (if there is one).
<code>_NonMDI</code>	Creates the form window outside of the MDI region. The window will always be on top of all MDI windows, and if moved outside of the CA-Realizer window, will not cause the MDI scroll bars to appear.
<code>_Palette</code>	A combination of <code>_NonMDI</code> + <code>_Title</code> + <code>_Size</code> + <code>_Close</code> + <code>_NoMax</code> .
<code>_PopUp</code>	Creates the form window completely outside of the CA-Realizer window.
<code>_Toolbar</code>	A combination of <code>_NonMDI</code> + <code>_Borderless</code> + <code>_HotClick</code> .

For example, the following lines create and display a form:

```
FormNew(10)  
FormControl(_Size; _Center, _Center, 75 pct, 75 pct)  
FormControl(_Show)
```



This form creates a default window style that resembles a dialog box in the following ways:

- It has no title or scroll bars.
- It cannot be closed.
- It is displayed at a fixed size and position.

Since there is no title bar on the default form, there is no way to close the form window with the mouse. To close the current form window, use the `FormControl(_Close)` command. To close all tool windows simultaneously, use the `RESET` command or `CTRL+ALT+F2`. For more information about the `FormControl` command, see *Controlling a Form* later in this chapter.

You can make a form window behave like other tool windows by adding the standard style constants to the `FormNew` command. Several examples are presented below.

Creating a Movable Form Window

The following example creates a movable form window by defining a title bar in the `FormNew` command with the `_Title` constant. Try running this example. Note that you can now move the form by dragging the title bar.

```
FormNew(20; "Movable Form", _Title)
FormControl(_Show)
```

Creating a Sizable Form Window

To create a movable and sizable form window, add both the `_Title` and `_Size` constants together as follows:

```
FormNew(30; "Movable Sizable Form", _Title + _Size)
FormControl(_Show)
```

To resize this form, drag a corner or border.

Creating a Closable Form Window

Add the `_Title` and `_Close` constants to make a form window closable:

```
FormNew(40; "Movable Closable", _Title + _Close)
FormControl(_Show)
```

To close this form, double-click on the system menu.

Now close the other forms with the `FormSelect` and `FormControl(_Close)` commands:

```
FormSelect(30)
FormControl(_Close)
FormSelect(20)
FormControl(_Close)
FormSelect(10)
FormControl(_Close)
```

Creating a Non-MDI Form Window

To create a non-MDI form, execute the following:

```
FormNew(10; "Non MDI Form", _Title + _NonMDI)
FormControl(_Show)
```

Note: Unlike an MDI window, if you drag the form outside of the CA-Realizer client area, scroll bars do not automatically appear. For more information about non-MDI window forms, see *Creating Toolbars, Status Bars, and Palettes Using Non-MDI Forms* later in this chapter.

Controlling a Form

When a form is created, it becomes the current form to which subsequent form commands apply. The following commands and functions allow you to manipulate forms.

Selecting a Form

To select a different form to be the current form, use the `FormSelect` command.

```
FormSelect (FormNum)  
FormNum = FormQ(_Selected)
```

Controlling How a Form Appears

The `FormControl` command allows you to control the appearance of the currently selected form.

```
FormControl (Action [, [StyleFlags,] [Left, Top  
[, Width, Height]])  
FormControl (Action [, [Style,] [MinWid, MinHt  
[, MaxWid, MaxHt]])
```

The following table lists the valid values for *Action*:

Action	Description
_Close	Closes the form, thereby destroying it.
_Hide	Hides the form.
_Maximize	Displays the form filling the entire CA-Realizer window.
_Minimize	Minimizes the form to an icon.
_NoFooter	Prevents the page number from being printed at the bottom of the page.
_NoFrame	Prevents the border of the form from being printed.
_NoHeader	Prevents a header from being printed at the top of the page.

Continued

Continued

Action	Description
<code>_Print</code>	Prints the form. The size and position of the form on the page can be specified with <i>Left</i> , <i>Top</i> , <i>Width</i> , and <i>Height</i> using the CA-Realizer position notation. (For more information about position notation, see “Overview of the CA-Realizer Programmable Application Tools.”) If <i>Width</i> and <i>Height</i> are used, the form is scaled as a bitmap. The printer style flags can be set with <i>StyleFlags</i> .
<code>_Restore</code>	Restores a maximized or minimized form to its previous size.
<code>_SetDrop</code>	If <i>Style</i> is 1, the form accepts drag and drop objects onto the form background. When this occurs, the form procedure is notified with <i>params[_Invoke]</i> set to <code>_DragNDrop</code> and <i>params[_ItemNum]</i> set to 0. If <i>Style</i> is 0, the form does not accept drag and drop objects. For more information, see Drag and Drop later in this chapter.
<code>_SetMode</code>	Changes the mode of a form, and in the case of modeless forms, changes how it responds to actions. If <i>Style</i> is set to <code>_Pick</code> , any click in the form causes the form procedure to return the object ID for the object that was clicked. <code>_PickDrag</code> mode is similar, except that the user can drag the object to another position in the form. Setting <i>Style</i> to <code>_Normal</code> returns the form to its normal state. For more information, see Modeless Pick and Drag later in this chapter.

Continued

Continued

Action	Description
<code>_SetFont</code>	Changes the default font for the form to the font specified by <i>Style</i> . This font is used for newly created objects, and for positionings based on line and char position units. If <i>Style</i> is 0, the default font is used.
<code>_Show</code>	Displays the form and makes it the active window.
<code>_Size</code>	Sets the size and the position of the form according to the size modifiers: <i>Left</i> , <i>Top</i> , <i>Width</i> , and <i>Height</i> . These values are specified using the CA-Realizer position notation. (For more information about position notation, see "Overview of the CA-Realizer Programmable Application Tools.")
<code>_SizeInside</code>	Sets the size and the position of the client area of the form according to the size modifiers: <i>Left</i> , <i>Top</i> , <i>Width</i> , and <i>Height</i> . These values are specified using the CA-Realizer position notation. (For more information about position notation, see "Overview of the CA-Realizer Programmable Application Tools.")
<code>_SizeTrack</code>	Specifies a minimum and maximum size for the form window according to the size modifiers: <i>MinWid</i> , <i>MinHt</i> , <i>MaxWid</i> , <i>MaxHt</i> . All values are specified in pixels.

Obtaining Information About a Form

To obtain information about an existing form, use the `FormQ` function:

```
ReturnVal = FormQ(Action [, FormNum])
```

The following table lists the valid values for *Action*:

Action	Description
<code>_Exists</code>	Returns 1 if the form exists; otherwise, it returns 0.
<code>_HWnd</code>	Returns the window handle of the form for use with external procedures and custom controls.
<code>_Selected</code>	Returns the ID of the currently selected form.
<code>_Size</code>	Returns a four-element array representing the size and position of the form in pixels. The first element is the location of the left side of the form, the second is the location of the top of the form, the third is the width of the form, and the fourth is the height of the form.
<code>_SizeInside</code>	Returns a four-element array representing the size and position of the client region of the form in pixels. The first element is the location of the left side of the client region, the second is the location of the top of the client region, the third is the width of the client region, and the fourth is the height of the client region.
<code>_Style</code>	Returns the style of the window, as set by <code>FormNew</code> .
<code>_Title</code>	Returns the title of the form.

Setting the Colors in a Form

Color is an important aspect of application design that allows you to communicate relationships and unify elements in an application.

Forms can display objects in many colors. You can choose an object's color before or after you create it, or you can modify the color of an existing object.

The `FormSetColor` command—which applies to the currently selected form—allows you to specify the color either by specifying RGB values ranging from 0 to 1 for each of the primary RGB colors, or by specifying one of CA-Realizer's predefined colors listed below:

<code>_Black</code>	<code>_DarkPurple</code>	<code>_LimeGreen</code>	<code>_PowderBlue</code>
<code>_Blue</code>	<code>_Evergreen</code>	<code>_GreenYellow</code>	<code>_DarkBrown</code>
<code>_Tan</code>	<code>_Gray</code>	<code>_Maroon</code>	<code>_DarkGreen</code>
<code>_Brick</code>	<code>_GrayGreen</code>	<code>_MediumBlue</code>	<code>_Turquoise</code>
<code>_Brown</code>	<code>_Green</code>	<code>_BlueGreen</code>	<code>_MediumBrown</code>
<code>_Cream</code>	<code>_Pale</code>	<code>_MediumGreen</code>	<code>_DarkGreenBlue</code>
<code>_Cyan</code>	<code>_Magenta</code>	<code>_NavyBlue</code>	<code>_LightEvergreen</code>
<code>_Purple</code>	<code>_LightGray</code>	<code>_Orange</code>	<code>_Yellow</code>
<code>_Red</code>	<code>_Pink</code>	<code>_LightYellow</code>	
<code>_Violet</code>	<code>_White</code>	<code>_LightGreen</code>	

The general form of the command is:

```
FormSetColor(Color [; Which])
```

```
FormSetColor(Red, Green, Blue [; Which])
```

Which specifies whether the color is for the form's background (`_Background`), the form object's text (`_Text`), or the form object's background (`_Field`).

Color changes affect only those objects created *after* the colors are set—existing objects are not affected, unless the `FormModifyObject(_SetColor)` command is used.

Setting the Background Color

The forms created thus far all use the default background color. You can change the color of the form with the `FormSetColor` command:

```
FormSetColor(_Red; _Background)
```

CA-Realizer updates the form color as soon as it executes the `FormSetColor` command.

Setting the Color of an Object

The `FormSetColor` command also applies to objects within the form.

For instance, to change the color of text found in objects like captions and edit controls, use the `_Text` modifier with `FormSetColor`:

```
FormSetColor(_Blue; _Text)
```

When CA-Realizer adds new objects to the form after this command has been executed, they will have blue text.

Objects such as option buttons, check boxes, group boxes, and edit controls have their own colored background field as well. To set the color of an object's field, use the `_Field` modifier with the `FormSetColor` command:

```
FormSetColor(_Red; _Field)
```

When CA-Realizer adds these objects to the form, they will have a red background color.

Adding Form Objects

A form object is any application tool (for example, a chart or a spreadsheet) or interface control (such as a check box, button, or list box) contained in a form. Every form object is identified with a unique object number (ID)—assigned when you create it. As with the application tools, you always reference objects by their ID.

Use the object's ID to identify the object when the user interacts with it. Object IDs 1 and 2 are automatically attached to the ENTER and ESC keys, so buttons or controls with these IDs should correspond to the actions associated with these keys. For instance, an OK button is often given an ID of 1, and a CANCEL button is often ID 2.

The major application tools—charts, sheets, logs, and boards—can all exist as stand-alone tools from a form. However, when you place them in a form, they become form objects—you create them differently, but their behavior remains the same.

There are also two other tools, graphics tablets and animation windows, that can *only* be used in a form.

Each form object behaves in its own characteristic way. Objects like spreadsheets, buttons, and check boxes react to a mouse click. Other objects, such as logs, combo boxes, and edit controls, accept input from the keyboard.

Using the FormSetObject Command

To place objects in a form once it has been created, use the FormSetObject command:

```
FormSetObject (ID, Type, Text [, Font],
              Left, Top [, Width, Height]
              [, [Style, ] Mod1 [, Mod2, ..., ModN]])
```

The FormSetObject command requires at least five parameters. The first parameter, *ID*, is an object number unique to that form. You should always reference form objects by their ID.

The second parameter, *Type*, is a constant describing the object type.

The following table lists the various *Type* constants:

Type	Description
_Animate	Animation window
_Bitmap	Bitmap file
_BitmapButton	Bitmap button
_Board	Board window
_Button	Push button
_CaptionCenter	Centered caption
_CaptionLeft	Left-justified caption
_CaptionRight	Right-justified caption
_Chart	Chart window
_CheckBitmap	Bitmap check box
_CheckBox	Check box
_ComboBox	Combo list box
_DefButton	Default push button
_DigitalClock	Digital clock

Continued

Continued

Type	Description
_DropDownCombo	Drop-down combo list box
_DropDownList	Drop-down list box
_EventTimer	Countdown timer
_Frame	Sizable frame
_GroupBox	Group box
_ListBox	List box
_Log	Log window
_Metafile	Metafile picture
_OLE	OLE client window
_OptionBitmap	Bitmap option button
_OptionButton	Option button
_PCX	PCX picture file
_Picture	Picture resource
_ScrollBar	Scroll bar
_Sheet	Sheet window
_Tablet	Tablet window
_TextBox	Edit control

The third parameter in `FormSetObject`, *Text*, is a text string. This string supplies information unique to the type of object being created. For example, it can specify the caption that is displayed within a button, or it can specify the name of a bitmap file to be displayed in the form.

FormSetObject also permits you to specify accelerator keys for form objects. An accelerator is a keystroke combination that allows you to select a form object using the keyboard (rather than with the mouse or by tabbing through other options). If the *Text* parameter contains an ampersand (&), the letter following the ampersand is underlined and the ALT+*char* combination is an active accelerator for that object. For example,

```
FormNew  
FormSetObject(3, _Button, "I&tems", 5,5)  
FormControl(_Show)
```

Note that you can include a literal ampersand in *Text* by using a double ampersand (&&).

For more information about accelerator keys, see Adding Accelerator Keys later in this chapter.

You must always specify the two required position parameters, *Left* and *Top*, which describe the object's horizontal and vertical position relative to the upper-left corner of the form window.

You can also include the optional *Font*, *Width*, and *Height* parameters with the FormSetObject command. Supply a defined font ID for *Font* to specify the font in which an object's text should be displayed. For example, you might want to display a caption in a certain font, size, and style. If *Font* is not specified, the default font is used. The default font can be with the FormControl(_SetFont) command.

Additionally, you can use *Width* and *Height* to set the size of an object using the CA-Realizer position notation.

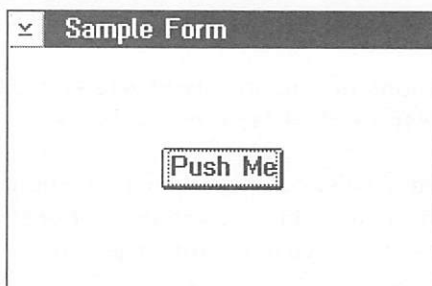
You can add several modifiers to FormSetObject, specifying the style, state, or contents of a form object. The modifier depends upon the form object.

For example, the following commands define a button labeled PUSH ME, identified by the object ID 10. The button is centered both horizontally and vertically:

```
FormNew(10; "Sample Form", _Title + _Close)
FormControl(_Size; _Center, _Center, 40 pct, 40 pct)
FormSetObject(10, _Button, "Push Me", _Center, \
    _Center)
FormControl(_Show)
```

To set the size of the button, you can add the *Width* and *Height* parameters to the `FormSetObject` command:

```
FormSetObject(10, _Button, "Push Me", _Center, \
    _Center, 30 pct, 20 pct)
```



You have just created a form with a single button. When you choose the button, it depresses. This means CA-Realizer is registering the mouse click. The button notifies the form immediately *after* you release the mouse.

In order for your application to be able to respond to notification, you must write the code to process the manipulations and select an appropriate response. For more information, see *Form Processing* later in this chapter.

Adding Buttons

The `_Button` and `_DefButton` constants can be used to create buttons and default buttons:

```
FormSetObject(ID, _Button, Text [, Font ],
              Left, Top [, Width, Height])

FormSetObject(ID, _DefButton, Text [, Font ],
              Left, Top [, Width, Height])
```

The text in *Text* is displayed in the button. If *Font* is not specified, the default font is used. Buttons can be any size.

For example, the following statement would create a default button that displays the text “OK”:

```
FormSetObject(1, _DefButton, "OK", 30 px1, 30 px1)
```

The standard usage of a default button indicates what will happen if the ENTER key is pressed. Therefore, it is misleading to create a default button with an ID other than 1 or to have more than one default button in a form.

Adding Option Buttons

Option buttons can be created with the `_OptionButton` constant:

```
FormSetObject(ID, _OptionButton, Text
              [, Font ], Left, Top [, Width, Height]
              [, [Style, ] InitState [, GroupNum]])
```

Use *Text* and *Font* to create the text for the button. If *Font* is not specified, the default font is used. If *Width* and *Height* are not specified, or if you specify `_Default` for *Width* and *Height*, the system default widths and heights will be used. The use of default widths and heights is recommended to ensure that enough space is allocated for the object’s display.

By default, the button is off when it is created. You can specify the initial state with *InitState*—use 0 for off and 1 for on.

Option buttons usually occur in groups. To group option buttons together, use *GroupNum*, giving each button in the group the same group number. Only one option button within a group can be on at one time.

As an example, the following statements could be used to create the option buttons "Mr." and "Mrs." for *GroupNum* 15. The initial state of "Mr." is set on:

```
FormSetObject(11, _OptionButton, "Mr.", \\  
    30 pxl, 30 pxl; 1, 15)  
FormSetObject(12, _OptionButton, "Mrs.", \\  
    30 pxl, 60 pxl; 0, 15)
```

By default, option buttons do not notify the program when they are turned on or off. It is usually easier to poll their status when the user has finished interacting with the form. If notification of every option button selection is needed, specify *Style* as *_Notify*.

Adding Bitmap Option Buttons

Use the *_OptionBitmap* constant to create a bitmap option button:

```
FormSetObject(ID, _OptionBitmap, Text,  
    Left, Top [, Width, Height]  
    [, [Style, ] InitState [, GroupNum]])
```

Bitmap option buttons function exactly the same as regular option buttons, except that *Text* is the name of a bitmap file containing the images to use for the on and off states of the button.

The bitmap is automatically cut in half. The left half is used when the option button is on, and the right half is used when the option button is off.

Adding Check Boxes

To create a check box, use the `_CheckBox` constant:

```
FormSetObject(ID, _CheckBox, Text [, Font ],  
             Left, Top [, Width, Height]  
             [, [Style, ] InitState])
```

Text is the text displayed for the check box, and *Font* is the font in which to display *Text*. If *Font* is not specified, the default font is used. If *Width* and *Height* are not specified, or if you specify `_Default` for *Width* and *Height*, the system default widths and heights will be used. The use of default widths and heights is recommended to ensure that enough space is allocated for the object's display.

By default, the check box is off when it is created. You can specify the initial state with *InitState*—use 0 for off and 1 for on.

For example, you might create the following check box:

```
FormSetObject(14, _CheckBox, "Winter", 30, 50)
```

By default, check boxes do not notify the program when they are turned on or off. It is usually easier to poll their status when the user has finished interacting with the form. If notification of every change is needed, specify *Style* as `_Notify`.

Adding Bitmap Check Boxes

Use the `_CheckBitmap` constant to create a bitmap check box:

```
FormSetObject(ID, _CheckBitmap, Text,  
             Left, Top [, Width, Height][, [Style, ]  
             InitState [, GroupNum]])
```

Bitmap check boxes function exactly the same as regular check boxes, except that *Text* is the name of a bitmap file containing the images to use for the on and off states of the check box.

The bitmap is automatically cut in half. The left half is used when the check box is on, and the right half is used when the check box is off.

Adding Group Boxes

You can create a group box with the `_GroupBox` constant:

```
FormSetObject(ID, _GroupBox, Text [, Font ],  
             Left, Top [, Width, Height]  
             [, Style [, _AccelNotify, NotifyObj]])
```

Use *Text* and *Font* to add a title to the group box in a specified font. If *Font* is not specified, the default font is used.

Note that the interior of a group box is not transparent. Therefore, you should create the group box before the objects that are to be placed inside it or else they will be obscured by the box itself.

To create group boxes that notify the form when they are clicked, set *Style* to `_Notify`. The `_AccelNotify` flag allows the group box to notify object *NotifyObj* if the object's accelerator key is used. (Accelerators are described later in this chapter.)

Note: You can create solid colored rectangles by setting the field color and creating a group box with an empty string for the text.

Adding Edit Controls

Use the `_TextBox` constant to create edit controls:

```
FormSetObject(ID, _TextBox, Text [, Font ],  
             Left, Top [, Width, Height] [, Style])
```

You can set the initial text inside the box with *Text*, and you can set the font with *Font*. If *Font* is not specified, the default font is used. A default height is recommended.

An optional modifier, *Style*, can be used to specify the style of the edit control. By default, edit controls are a single line and have a border. The `_MultiLine` and `_NoBorder` flags can be used or added together as *Style*.

As an example, the following statement would create an edit control with its initial text set to "January":

```
FormSetObject(16, _TextBox, "January", 30, 30)
```

To verify that the data a user enters into edit controls on a field-by-field basis is valid, add the `_Notify` flag to *Style*. For example, a program may want to ensure that the data entered in a date-of-birth field is in fact an actual date. If the `_Notify` flag is added to *Style*, the program is notified when the user leaves an edit control.

Adding Captions

To create captions, use the `_CaptionLeft`, `_CaptionRight`, and `_CaptionCenter` constants:

```
FormSetObject(ID, _CaptionLeft, Text
    [, Font ], Left, Top [, Width, Height]
    [, Style] [, _AccelNotify, NotifyObj])

FormSetObject(ID, _CaptionRight, Text
    [, Font ], Left, Top [, Width, Height]
    [, Style] [, _AccelNotify, NotifyObj])

FormSetObject(ID, _CaptionCenter, Text
    [, Font ], Left, Top [, Width, Height]
    [, Style] [, _AccelNotify, NotifyObj])
```

Use *Text* and *Font* to add a caption in a specified font. If *Font* is not specified, the default font is used.

If the caption is only a single line, you should set the *Height* to `_Default`. You can create multiline captions by using carriage returns (ASCII 13) in *Text*.

To create captions that notify the form when they are clicked, set *Style* to `_Notify`. The `_AccelNotify` flag allows the caption to notify object *NotifyObj* if the object's accelerator key is used.

Adding Pictures

Graphic images can be added to the form in a number of ways.

Bitmap and Metafile To add bitmaps and metafiles, use the `_Bitmap` and `_Metafile` parameters:

```
FormSetObject(ID, _Bitmap, Bitmap, Left, Top  
[, Width, Height])
```

```
FormSetObject(ID, _Metafile, Metafile, Left,  
Top [, Width, Height])
```

Bitmap Option Button To add bitmap option buttons, use the `_BitmapButton` parameters:

```
FormSetObject(ID, _BitmapButton, Bitmap, Left,  
Top [, Width, Height])
```

Bitmap or *Metafile* is either a file name or a picture ID.

PCX File To add PCX files, use the `_PCX` constant:

```
FormSetObject(ID, _PCX, FileName, Left, Top  
[, Width, Height])
```

FileName indicates the name of the PCX file.

Picture Resource To add a picture resource, use the `_Picture` constant:

```
FormSetObject(ID, _Picture, PicNum, Left,  
Top [, Width, Height])
```

If you do not fully specify a file name, CA-Realizer searches the `_LoadDir` path. If no file extension is specified, the proper default is assumed.

If you specify *Width* and *Height*, the image is rescaled to those values. If the default is used, the image is drawn as its actual size.

Adding List Boxes

You can create four list box types using the `_ListBox`, `_ComboBox`, `_DropDownList`, and `_DropDownCombo` constants:

```
FormSetObject(ID, _ListBox, "" [, Font],
              Left, Top [, Width, Height] [, Style, ]
              Mod1 [..., ModN])

FormSetObject(ID, _ComboBox, "" [, Font],
              Left, Top [, Width, Height]
              [, Style, ] Mod1 [..., ModN])

FormSetObject(ID, _DropDownList, "" [, Font],
              Left, Top [, Width, Height]
              [, Style, ] Mod1 [..., ModN])

FormSetObject(ID, _DropDownCombo, "" [, Font],
              Left, Top [, Width, Height]
              [, Style, ] Mod1 [..., ModN])
```

The *Text* field is ignored—this is indicated in the syntax as an empty string (`""`).

For example, the following statement creates a combo box:

```
FormSetObject(19, _ComboBox, "", 30, 30)
```

There are a wide variety of modifiers that you can use to describe the contents and behavior of the list boxes. For example, if the contents of the list box should be kept sorted, specify `_Sorted` as part of the *Style* modifier. A regular `_ListBox` can also include the `_MultipleSel` style to create a multiple-selection list box.

By default, a regular `_ListBox` notifies the form with a `_Click` message when the user double-clicks an item. The other list boxes do not. If the `_Notify` flag is added to *Style* for the other list boxes, they notify the form with a `_Click` message when a new item is selected. If `_Notify` is used with `_ListBox`, a `_Change` message is sent every time a new item is selected.

The rest of the modifiers come in groups. Each group adds a set of items to the list box. Any number of modifier groups can be used together.

Adding File Names to a List Box

You can fill a list box with the file names from a directory using the `_ListFiles` modifier as follows:

```
FormSetObject(ID, ListBoxType ... ; \\
... _ListFiles, Flags, Filter ... )
```

Flags controls the files and directories that are included. It is the sum of any of the following: `_NormFiles`, `_Drives`, `_SubDirs`, and `_NoFiles`. The default value is `_NormFiles`.

Filter is a file name filter string, such as `"*.RLZ"` or `"ABC???.TXT"`. CA-Realizer uses the current directory unless the filter begins with a path specification, such as `"C:\WINDOWS\*.INI"`.

Adding Variable Names to a List Box

Use the `_ListVars` modifier to fill a list box with a list of the active CA-Realizer variable names:

```
FormSetObject(ID, ListBoxType ... ; \\
... _ListVars, Flags ... )
```

Flags indicates the variables to place in the list. It is the sum of any of the following: `_Alpha`, `_Real`, `_DateTime`, `_Scalar`, and `_Array`. For example, `_Real + _DateTime` produces a list of all the numeric and date-time variables, whether they are scalar or array. `_Real + _DateTime + _Scalar` produces a smaller list of all the numeric and date-time scalars.

Adding Family Names to a List Box

Use the `_ListFams` modifier to create a list of the names of all the currently defined CA-Realizer families:

```
FormSetObject(ID, ListBoxType ... ;
... _ListFams ... )
```

Adding Font Names to a List Box

You can use either the `_ListFonts` modifier or the `_ListPrFonts` modifier to populate a list box with the names of the available system or printer fonts, respectively:

```
FormSetObject(ID, ListBoxType ... ;  
... _ListFonts ... )  
  
FormSetObject(ID, ListBoxType ... ;  
... _ListPrFonts ... )
```

Adding Font Sizes to a List Box

To populate a list box with the available font sizes for a specific system or printer font, use the `_ListFontSizes` or `_ListPrFontSizes` modifier, respectively:

```
FormSetObject(ID, ListBoxType ... ;  
... _ListFontSizes, FontName ... )  
  
FormSetObject(ID, ListBoxType ... ;  
... _ListPrFontSizes, FontName ... )
```

FontName is the name of either the system or printer font whose available sizes you want to list.

Adding Text to a List Box

You can put a list of strings in a list box by listing them as the modifiers. You can use any combination of string scalars or arrays. The syntax is presented below:

```
FormSetObject(ID, ListBoxType ... ;  
... Text ... )
```

Combining Modifier Groups

You can combine groups of modifiers to create list boxes with varied information. Two examples are presented below:

```
'Create a list of the names of all real scalars and
'string arrays
FormSetObject(ID, _ListBox ... ; _ListVars, \
    _Real+_Scalar, _ListVars, _Alpha+_Array)

'Create a list of all the fonts and one extra entry
FormSetObject(ID, _ListBox ... ; _ListFonts, \
    "New Font")
```

Adding Frames

Use the `_Frame` type constant to create frames:

```
FormSetObject(ID, _Frame, Text, [Font, ]
    Left, Top [, Width, Height])
```

Text is kept centered in the frame as the user resizes it.

Adding Scroll Bars

Scroll bars are created by using the `_ScrollBar` type constant:

```
FormSetObject(ID, _ScrollBar, "", Left, Top
    [, Width, Height] [, [Style, ] Value
    [, Min [, Max [, PageSize]]])
```

Text is ignored—this is indicated in the syntax as an empty string (`""`). The *Value* modifier sets the initial position of the scroll box. It must be between the minimum and maximum values of the scroll bar, set by *Min* and *Max*. By default, the scroll bar ranges from 0 to 100 with an initial value of 0.

The *PageSize* modifier indicates how far the scroll box moves if the user clicks within the scroll bar. For example, if *PageSize* is 40 when the scroll bar ranges from 1 to 200, then the scroll box will move 40 units (20%) when the user clicks in the scroll bar.

If *PageSize* is negative, it represents a percentage of the scroll bar. For example, -15 sets the page size to 15%. The default page size is 10%.

The optional *Style* modifier describes the type of scroll bar to create. It can be any of the flags shown in the following table, added together with the `_SB_Style` constant:

Style	Effect
<code>_SB_Vertical</code>	Creates a vertical scroll bar.
<code>_SB_Horizontal</code>	Creates a horizontal scroll bar.
<code>_SB_AlignAlone</code>	Adjusts automatically to fill the full width or height of the form.
<code>_SB_AlignWithOther</code>	Adjusts automatically to fill the full width or height of the form, but leaves enough room for the opposite scroll bar.

If either of the alignment parameters is used, the scroll bars will automatically be positioned on the right or bottom of the form. Both size and position parameters will be ignored.

For example, the following statements create a centered horizontal scroll bar:

```
FormSetObject(10, _ScrollBar, "", _Center, _Center;  
              _SB_Style + _SB_Horizontal)
```

Adding Digital Clocks

To add a digital clock to a form, use the `_DigitalClock` type constant:

```
FormSetObject(ID, _DigitalClock, Format "[, Font ]  
            [, Left, Top [, Width, Height])
```

The format parameter is a format string such as those used by `PRINT USING` indicating the format for the time. For example, `"D(h1;m1;s1)"` displays the time in a 24-hour format (for example, 15:45:00). `"D(h4;m1 h5)"` displays as 3:45 PM, and `"D(M1\D1\Y1)"` displays the date. The default, an empty string the time in 24 hour-format.

Adding Event Timers

Countdown event timers can be added to the form by using the `_EventTimer` type constant:

```
FormSetObject(ID, _EventTimer, Text  
            [, Font ], Left, Top[, Width, Height]  
            [, Count [, Format]])
```

Text is ignored—this is indicated in the syntax as an empty string (`""`). *Count* is the initial value of the countdown timer. By default, it is 0. *Format* specifies the format for the timer's display.

The available formats are listed below:

<i>Format</i>	<i>Format</i>	<i>Example</i>
<code>_ET_Sec</code>	ssss	125
<code>_ET_MinSec</code>	mm:ss	02:05
<code>_ET_HourMinSec</code>	hh:mm:ss	01:45:20

Adding OLE Objects

Linked objects from other applications can be added by using the `_OLE` type constant:

```
FormSetObject(ID, _OLE, Object, Left, Top
[, Width, Height])
```

Object is the file name of the object. The system uses the file extension to determine the proper server application to use.

Adding CA-Realizer Tools

The CA-Realizer tools are placed in a form with the `_Chart`, `_Sheet`, `_Board`, `_Log`, `_Tablet`, and `_Animate` type constants as follows:

```
FormSetObject(ID, _Chart, "", Left, Top
[, Width, Height])
```

```
FormSetObject(ID, _Sheet, "", Left, Top
[, Width, Height])
```

```
FormSetObject(ID, _Board, "", Left, Top
[, Width, Height]; Template
[, Data1 [..., DataN]])
```

```
FormSetObject(ID, _Log, "" [, Font ],
Left, Top [, Width, Height])
```

```
FormSetObject(ID, _Tablet, "", Left, Top
[, Width, Height])
```

```
FormSetObject(ID, _Animate, "", Left, Top
[, Width, Height])
```

CA-Realizer ignores *Text* and *Font* parameters for all tools except logs.

You cannot put variables into a sheet when it is created. Instead, add variables with the SheetUpdate command.

Positioning Objects in a Form

When objects are placed in a form, the position values are specified using the position notation. The possible units include pixels, percents, inches, millimeters, points, twips, lines, and characters. (For more information about CA-Realizer's position notation, refer to the "Overview of the CA-Realizer Programmable Application Tools" chapter in this guide.)

Certain position units are better suited for creating device- and resolution-independent forms. If pixels are used, the objects will be smaller on higher resolution monitors. If percentages are used, the objects will have the same relative sizes and spacings regardless of resolution.

The lines and characters units provide another level of device independence. With these units, a coordinate system is created based on the character height and the average character width of a specific font. By default, the CA-Realizer system font is used. The font can be changed with the FormControl(_SetFont) command.

Manipulating and Modifying Form Objects

You can use the `FormModifyObject` command to modify the state and appearance of any object once you have created it in a form. This command allows you to:

- Change most of the object attributes set by the `FormSetObject` command
- Change the state of an object
- Disable an object
- Add new attributes to an object

The general format for the `FormModifyObject` command is as follows:

```
FormModifyObject(ID [, Attribute [, Text]  
                [, Left, Top [, Width, Height]]]  
                [; Mod1 [ .., ModN]])
```

You will recognize some of these parameters from the `FormSetObject` command. For instance, the *Text*, *Left*, and *Top* parameters change the appearance and location of an object.

The *Attribute* parameter, on the other hand, is unique to the *FormModifyObject* command. There are several attributes that you can specify with this parameter. The following table lists the valid *Attribute* values:

<i>Attribute</i>	<i>Description</i>
<i>_BringToFront</i>	Adjusts the front-to-back and tab-stop ordering of the objects in the form so that the object specified in the <i>FormModifyObject</i> command as <i>ID</i> is the topmost and first object.
<i>_Clear</i>	Clears the contents of any list box type.
<i>_Close</i>	Removes the specified object from the form.
<i>_Drag</i>	Sets the specified object as draggable.
<i>_DragOff</i>	Sets the specified object as non-draggable.
<i>_Drop</i>	Sets the specified object as a drop receiver.
<i>_DropOff</i>	Sets the specified object as <i>not</i> a drop receiver.
<i>_Gray</i>	Grays and disables the specified object.
<i>_Hide</i>	Hides the specified object.
<i>_Normal</i>	Restores the specified object to its normal (ungray) state.
<i>_SendBehind</i>	Adjusts the front-to-back and tab-stop ordering of the objects in the form so that the object specified in the <i>FormModifyObject</i> command as <i>ID</i> , is directly after the object specified by <i>Mod1</i> . <i>Mod1</i> can also be <i>_All</i> (the default) or <i>_None</i> .
<i>_SetColor</i>	Sets the colors of the specified object to the currently selected colors.

Continued

Continued

Attribute	Description
<code>_SetFocus</code>	Forces the focus to be set to the specified object. The object with focus is normally shown with a dotted gray border around it.
<code>_Show</code>	Shows the specified object.
<code>_TabStop</code>	Controls whether the specified object is part of the tab-stop ordering. If <i>Mod1</i> is 0, the object will be removed from the tab list. If <i>Mod1</i> is 1, the object will be listed in the tab list.

The parameters and modifiers that you supply to both the `FormSetObject` and the `FormModifyObject` commands depend on the type of object that you are creating or modifying.

Closing an Object

To close a form object and remove it from a form, use the `FormModifyObject` command and set *Attribute* to `_Close`. For example, the following statement removes form object number 10:

```
FormModifyObject(10, _Close)
```

Changing the State of an Object

You can gray (or disable) form objects at any time by using the `FormModifyObject` command with an *Attribute* value of `_Gray` as follows:

```
FormModifyObject(ID, _Gray [...])
```

This prevents objects from responding to mouse input, and depending on the object, may change their appearance. For example, grayed buttons will not respond to mouse clicks and grayed edit controls cannot be edited.

Return form objects to their normal state by using `FormModifyObject` with an *Attribute* value of `_Normal` as follows:

```
FormModifyObject (ID, _Normal [...])
```

Note: Group boxes and captions are grayed by default when they are created, unless they were created with the `_Notify Style` flag.

Changing the Color of an Existing Object

Once an object has been created, you can change its colors to those that have been set by `FormSetColor` by using the `FormModifyObject` command with an *Attribute* value of `_SetColor`. For example, the following statement sets the color for object number 10 to the currently selected colors:

```
FormModifyObject (10, _SetColor)
```

Setting the Focus to an Object

Each form has one item that has focus (that is, is currently selected by the user). When the form is the frontmost window, the selected object is normally shown with a dotted gray border around it. This is called the *focus item*. The focus item is the item that currently receives keyboard input.

You can force the focus to a certain item by using the `FormModifyObject` command with an *Attribute* value of `_SetFocus`. For example, the following statement makes object number 10 the currently selected object:

```
FormModifyObject (10 , _SetFocus [...])
```

Adjusting the Front-to-Back and Tab-Stop Ordering of Objects

There is an implicit ordering of objects that is used to determine how overlapping objects are displayed and the order in which objects are visited when the user presses the TAB key. This front-to-back ordering is created as objects are added to the form. If objects overlap, the one that was created later is drawn on top. The tab-stop ordering is the same as the creation order.

This front-to-back ordering can be modified at any time. Use `FormModifyObject` with an *Attribute* value of `_SendBehind` to move an object behind another (and therefore make it earlier in the tab-stop ordering):

```
FormModifyObject(ID, _SendBehind [...]  
                [; BehindID])
```

BehindID is the object that it will directly precede. If *BehindID* is unspecified or `_All`, the object will be behind all others and first in the tab-stop ordering. If *BehindID* is `_None`, the object will be above all others and last in the tab-stop ordering.

Note: An *Attribute* value of `_BringToFront` is equivalent to `_SendBehind` with a *BehindID* of `_None`.

When an object is grayed, it will not be included in the tab stop list. To remove a non-grayed object from the tab stop list, use:

```
FormModifyObject(ID, _TabStop [...]; 0)
```

This disables its tab-stop position, but does not change the tab-stop order of the other objects. To reenable the tab stop, use:

```
FormModifyObject(ID, _TabStop [...]; 1)
```

Changing the Status of an Option Button or Check Box

Use the `FormModifyObject` command to turn option buttons or check boxes on or off:

```
FormModifyObject(ID [, ...] ; State)
```

Specify their new state with the *State* parameter, where the values 0 and 1 indicate off and on, respectively.

Modifying a List Box

Adding Items

You can add items to a list box by using the `FormModifyObject` command, and by specifying any of the modifier groups in the same way as with `FormSetObject`:

```
FormModifyObject(ID, ... ; Mod1 [... , ModN])
```

If the list box was not created with the `_Sorted` style, new items are added to the end of the list.

Clearing Contents

To clear the contents of a list box, use the `FormModifyObject` command with an *Attribute* value of `_Clear`:

```
FormModifyObject(ID, _Clear, ...)
```

Selecting Items Programmatically

Using the `_ListSelect` modifier with `FormModifyObject`, you can select items programmatically:

```
FormModifyObject(ID, ... ; _ListSelect, ItemNum)
```

ItemNum is the number of the item in the list box to select.

Deleting Items

You can delete an item in a list box by using the `_ListDelete` modifier with `FormModifyObject`:

```
FormModifyObject(ID, ... ; _ListDelete, ItemNum)
```

ItemNum is the number of the item to delete.

Inserting Items

Finally, you can use `FormModifyObject` and the `_ListInsert` modifier to insert items into the list box at a point other than the end:

```
FormModifyObject(ID, ... ;
    _ListInsert, ItemNum, Data1 [..., DataN])
```

ItemNum is the number of the item before which the new items will be placed. You can insert any combination of string scalars and arrays.

Note: All these commands work with any of the list box variations (`_ListBox`, `_ComboBox`, `_DropDownList`, and `_DropDownCombo`).

Modifying a Scroll Bar

The current value (*Value*) of the thumb control, as well as the range (*Min*, *Max*) and page size (*PageSize*) of the scroll bar, can be changed by specifying the new values for the appropriate modifiers with `FormModifyObject`:

```
FormModifyObject(ID [, ...] ; Value
    [, Min [, Max [, PageSize]]])
```

Modifying an Event Timer

The current value (*Count*) of the timer, as well as its format (*Format*), can be changed by specifying the new values for each with the `FormModifyObject` command as follows:

```
FormModifyObject(ID [, ...] ; Count
    [, Format])
```

If *Count* is 0, the timer is turned off.

Querying the Object

Use `FormQObject` to return an array of information about any item in a form:

```
ReturnArray = FormQObject (ID)
```

If *ID* is not specified, `FormQObject` returns information for the object that currently has focus. The following table lists the elements of the array returned from `FormQObject`:

Element	Description
<code>_FQO_ItemNum</code>	The ID of the object.
<code>_FQO_ItemType</code>	The type of item as set with <code>FormSetObject</code> .
<code>_FQO_Left</code>	The current position, in pixels, of the left side of the object.
<code>_FQO_Top</code>	The current position, in pixels, of the top of the object..
<code>_FQO_Width</code>	The width of the object in pixels.
<code>_FQO_Height</code>	The height of the object in pixels.
<code>_FQO_Style</code>	The style of the object (<code>_Normal</code> or <code>_Gray</code>).
<code>_FQO_Value</code>	The value of the object as would be returned by <code>FormQNum</code> .
<code>_FQO_HWnd</code>	The window handle of the object, for use with the <code>EXTERNAL</code> function.

The array may contain additional values which are for CA-Realizer only.

Obtaining the Front-to-Back Ordering

The current front-to-back and tab-stop ordering of the objects in the form can be obtained using the `_FQO_ItemList` constant with the `FormQObject` function as follows:

```
Order = FormQObject(_FQO_ItemList)
```

This command returns an array with the item numbers ordered from backmost to frontmost. If there are no objects in the form, an array with a single element of 0 is returned.

Querying the Object Value

Almost every form object has a value that is associated with it. This value is usually a number or a text string. For example, the numeric value of an option button or a check box indicates whether it is currently on or off, and the text value of an edit control is the string typed in by the user.

Once the user acts upon a form object, you can use two functions, `FormQNum` and `FormQStr`, to retrieve their current values. You apply the `FormQNum` and `FormQStr` functions to the object ID passed to them as a parameter.

Obtaining the Numeric Value of an Object with `FormQNum`

The `FormQNum` function returns the numeric value of a form object for the current form:

```
RetVal = FormQNum(ID)
```

ID is the object number of the item in the form.

For example, to obtain the number value from object number 10, you would use the following statement:

```
PRINT FormQNum(10)
```

Note: Although this function often appears within a `FormWait` loop or a form procedure, you can also poll form objects after form operations are complete.

Option Buttons and
Check Boxes

Option buttons, check boxes, bitmap option buttons, and bitmap check boxes return 1 or 0, indicating whether they are on or off, respectively.

List Boxes

All single-selection list boxes return the number of the currently selected item, or 0 if nothing is selected. Multiple-selection list boxes return 0 if there are no selections, or an array of the currently selected item numbers.

Event Timers

Event timers return the number of seconds remaining before the timer notifies the form.

Scroll Bars

Scroll bars return the current value of the thumb control.

Tip: The return value will be -2 if *ID* is invalid, and -1 for all other controls.

Obtaining the Text Value of an Object with `FormQStr`

The `FormQStr` function returns the text value of an object in the current form:

```
TextVal = FormQStr(ID [, Start [, Count]])
```

ID is the object number of the item in the form. *Start* and *Count* only apply to list box objects.

For example, to obtain the text values from a button created as object number 10, use the following statement:

```
PRINT FormQStr(10)
```

FormQStr returns the text value from the button name. Because a button has no numeric value, FormQNum would return a -1.

Note: Although this function often appears within a FormWait loop or a form procedure, you can also poll form objects after form operations are complete.

Edit Controls and Combo Boxes

For edit controls, combo boxes, and drop-down combo boxes, the returned string is the text within the edit field of the object.

List Boxes

For list boxes and drop-down list boxes, the string returned is the selected item, if any. For multiple-selection list boxes, the value returned is an array of strings containing all the selected items, or a scalar empty string if there are no items selected.

The contents of any list box can be retrieved as an array of strings by specifying *Start* and *Count*. *Start* indicates the first item to retrieve, and if *Count* is specified, only that many items will be returned. The valid range of the returned array will begin with *Start*.

If *Start* is not valid, the string array returned will be a single empty string with an index one greater than the actual number of elements in the list box. This can be used to find out how many items are actually in the list box.

Note: For all other objects, the string returned is the object's text as set by the FormSetObject command or the FormModifyObject command. If *ID* is invalid, an empty string will be returned.

Creating Toolbars, Status Bars, and Palettes Using Non-MDI Forms

By default, all forms are created as windows within the Multiple Document Interface (MDI) (or client) region. The MDI region is usually the entire CA-Realizer window.

MDI forms are automatically clipped within the MDI region, and if minimized, appear as icons at the bottom of the MDI region. If MDI forms are moved outside the MDI region, scroll bars appear on the right side or the bottom of the region to create a larger virtual window.

It is also possible to create forms that are outside the MDI region or even outside the CA-Realizer window itself. These types of forms are useful for creating toolbars, palettes, and pop-up windows.

Non-MDI Forms

To create a non-MDI form, add the `_NonMDI Style` flag to the other flags set when the form is created with the `FormNew` command as follows:

```
FormNew( ; "Non MDI Form", _Title + _NonMDI)
```

This form is not contained in the MDI region, nor will it force scroll bars to appear if it is moved outside the boundaries of the CA-Realizer window. Non-MDI forms always appear above MDI forms and windows.

Adjusting the MDI Region

The MDI region itself can be resized with `SetSys(_SizeMDI)`:

```
SetSys(_SizeMDI, Insets)
```

Insets is an array of four values indicating the number of pixels that the MDI region should be inset from the left, top, right, and bottom edges of the CA-Realizer window, respectively.

For example, the following statement leaves a 50-pixel region on the top for a toolbar and a 20-pixel region on the bottom for a status bar:

```
SetSys(_SizeMDI, {0, 50, 0, 20})
```

Resizing the MDI region leaves areas on the screen for non-MDI forms that remain in place while other forms move within the virtual MDI window.

Creating a Toolbar or Status Bar

A toolbar, in general, is a non-MDI form that appears at the top or side of the application window and contains status information and action buttons (often as bitmap option buttons and bitmap check boxes). Toolbar forms normally have no title or border, but appear to be simply a region of the screen.

Status bars are similar, except that they usually appear at the bottom of the screen and contain read-only status information.

The `_Toolbar Style` flag can be used alone in place of all other flags, including `_NonMDI`, when the form is created with `FormNew`. The `_Toolbar` style is equal to `_NonMDI + _Borderless + _HotClick`.

This example creates a 50-pixel-high gray toolbar at the top of the screen:

```
SetSys(_SizeMDI, {0, 50, 0, 0})
FormNew( ; "", _Toolbar)
FormSetColor(_LightGray; _Background)
FormControl(_Size; 0 pxl, 0 pxl, 100 pct, 50 pxl)
FormControl(_Show)
```

Creating a Palette

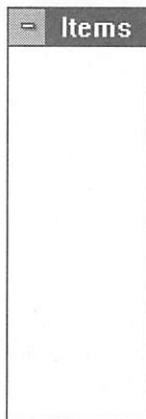
A palette, in general, is a movable—and possibly resizable—non-MDI form that contains a number of objects from which the user may choose. Palette objects are often shown as bitmap option buttons.

The palette window sits above all other windows, and will not bring up the MDI scroll bars if it is moved off the screen.

The `_Palette Style` flag can be used in place of all the other flags, including `_NonMDI`, when the form is created with `FormNew`. The `_Palette` style is equal to `_NonMDI + _Title + _Size + _Close + _NoMax`. Unwanted style elements, such as `_Size`, can be removed by subtracting them.

The following statements create a palette titled “Items” using the percent position notation:

```
FormNew( ; "Items", _Palette)  
FormControl(_Size; 10 pct, 10 pct, 10 pct, 50 pct)  
FormControl(_Show)
```



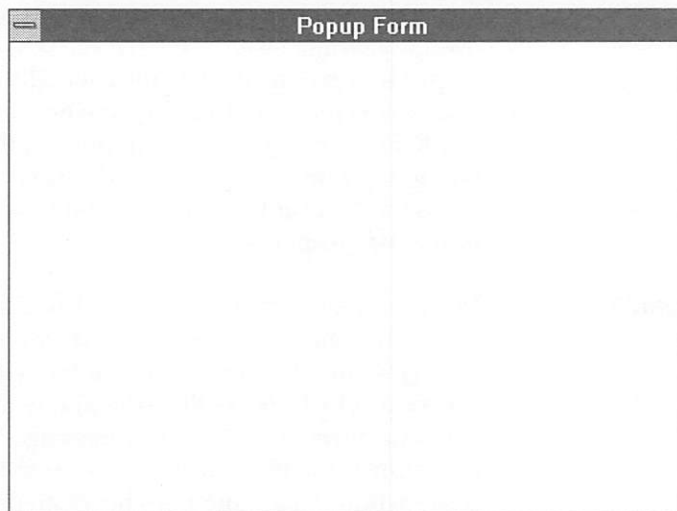
Creating a Pop-up Form

Pop-up forms are created outside the entire CA-Realizer application window. If the CA-Realizer window is resized or minimized, the pop-up will still remain in its position and will not be clipped.

Pop-up forms are created by specifying the `_Popup` style when the form is created with `FormNew`.

For example, the following statements create a pop-up form titled "Pop-up Form":

```
FormNew( ; "Popup Form", _Popup + _Title)  
FormControl(_Show)
```



Form Processing

Once you have created a form, it is displayed on the screen to allow the user to interact with it. You can handle this in two ways—by using modal or modeless forms.

Modal

Modal forms come to the front of the screen and wait for the user to complete an action before the form can be closed. While the user is waiting, other windows cannot be brought in front of the form. Modal forms are often used to handle questions that the user must answer (for example, “What file do you want to open?” or “Do you want to close this log?”) before continuing.

Modeless

The opposite of a modal form is a *modeless form*. Modeless forms can be interacted with, and then put aside while the user does something else. For instance, a CA-Realizer program that displays a text file and lets the user edit it should be designed as a modeless form. In this way, the user can run other CA-Realizer programs or manipulate other CA-Realizer windows at the same time. This form is more passive, since it waits for the user to interact with it instead of demanding an immediate response.

Using FormWait

With a modal form, the form is created and then the program explicitly waits for the user to perform some action(s) on the objects within the form. This is done with the `FormWait` function, which shows the form and waits for an action that will *notify* the form. For example, pressing a button notifies the form and returns the ID of the button so that the program can take some action. Changing the state of an option button, however, normally does not notify the form. The program usually waits until the user presses an OK button before checking the status of the option buttons to identify the one that was eventually selected. For more information about the `FormWait` function, see Modal Forms later in this chapter.

Modeless forms are notified of the same events as modal forms, but instead of explicitly asking which object was selected with a `FormWait`, a modeless form attaches a form procedure with the `FormSetProc` command. This procedure is called when a form object notifies the form, the same as with any other CA-Realizer tool. Form procedures are discussed in greater detail later in this chapter. For more information about tool procedures in general, see “Overview of the CA-Realizer Programmable Application Tools.”

Notification Messages

Most objects notify the form with a `_Click` message (although in some cases there is no mouse click involved). `_Click` messages cause `FormWait` to return a value, and will invoke the form procedure with an `_Invoke` value of `_Click`.

Modeless forms can also respond to other messages, such as right and middle mouse clicks. To temporarily prevent any object from notifying, you can use the `FormModifyObject` command to gray the object.

Button Notifications

Buttons and default buttons notify with a `_Click` message when they are pressed or clicked with the left mouse button, or when their accelerator key is used.

Buttons in modeless forms can also respond to right and middle mouse clicks.

Option Button and Check Box Notifications

Option buttons and check boxes—as well as bitmap option buttons and bitmap check boxes—normally do not notify the form. However, if they are created with the *_Notify Style* flag, they will notify with a *_Click* message whenever they are turned on or off. This can be useful if some other portion of the form needs to be updated when an option button or check box changes.

For example, a form that has an edit control for a person's height and a set of option buttons for the units (feet, inches, and centimeters) might want to convert the current value of the height whenever a different unit button is selected.

Edit Control Notifications

Edit controls do not notify the form by default. However, if the *_Notify Style* flag is used, they will notify with a *_Click* message when the user tries to leave the edit control by clicking elsewhere or by pressing ENTER, ESC, or the TAB key.

This can be used to ensure that the text is valid and to force the user to reenter the data if it is not. For more information, see the Edit Control Verification sections for both modal and modeless forms later in this chapter.

Group Box and Caption Notifications

By default, group boxes and captions do not notify the form. However, they will respond to left mouse button clicks and accelerator keys with a *_Click* message if they were created with the *_Notify Style* flag.

Group boxes and captions can also be set to notify a different object when their accelerator is activated if the *_AccelNotify* modifier is used when the object is created with the *FormSetObject* command.

Bitmap and Picture Notifications

Bitmaps and bitmap buttons notify with a `_Click` message when they are clicked with the left mouse button.

Modeless forms can determine the exact position of the click in the bitmap. For more information, see *Form Procedures* later in this chapter.

Frame Notifications

Frames notify the form with a `_Click` message when the user moves or resizes them.

List Box Notifications

Regular list boxes notify the form with a `_Click` message when an item is double-clicked. Other list boxes will notify only if they are created with the `_Notify Style` flag. In this case, notification occurs when the user changes the selection in the list portion.

If a regular list box is created with the `_Notify` flag in a modeless form, it will notify the form with a `_Change` message every time a new item is selected.

Event Timer Notifications

Event timers notify the form with a `_Click` message when they reach 0, and they continue to notify the form once every second until the timer is reset or turned off by setting its value to 0 with the `FormModifyObject` command.

Scroll Bar Notifications

Scroll bars notify the form with a `_Click` message when the user moves the thumb control in any way.

Graphic Tablet and Animation Notifications

Graphic tablets and animation windows notify the form with a `_Click` message when they are clicked. The position of the click can be determined by modeless form procedures. For more information, see Form Procedures later in this chapter.

Chart, Sheet, Board and Log Notifications

The other CA-Realizer tools (charts, sheets, boards, and logs) placed in a form do not notify the form—they notify their own tool procedure.

ENTER and Esc Notifications

The ENTER and ESC keys also notify the form with `_Click` messages, with IDs of 1 and 2, respectively. You cannot disable or gray these keys, although ENTER will not notify if the form was created with the `_ContextEnter Style` flag.

Modal Forms

A form becomes a modal form when you use the `FormWait` function:

```
ID = FormWait  
ID = FormWait(_Normal)
```

`FormWait` brings the current form to the front and waits until one of the objects notifies the form. During this time, the user can manipulate any of the objects in the form (for example, typing into an edit control, changing the status of an option button, or scrolling in a sheet), but cannot bring another window forward or manipulate the objects in a different form.

`FormWait` returns the ID of the object that notified the form with a `_Click` message, as described above. It is often used in a `LOOP` structure that continually waits for an object to notify and then does a `SELECT CASE` on the object ID. The processing for an object that indicates that the user is finished, such as pressing an `OK` or `CANCEL` button, will normally include an `EXIT LOOP` statement and a `FormControl(_Close)`.

Edit Control Verification

Edit controls are unique in the way that they notify the form. All other objects that can notify do so when the object is manipulated. If an edit control is created with the `_Notify` style option, it will notify the form when the user tries to leave it, either by clicking elsewhere or by pressing `ENTER`, `ESC`, or the `TAB` key.

You can use this notification to verify that the text that is entered in the edit control is legal for the field. For example, a field requesting a numeric value might require that the value be positive, a field for a date might check that the text is in fact an actual date value, and a string might have a limitation on its length.

The verification of an edit control can take place when it notifies the form. If the data is deemed illegal, a warning message can be displayed with the `INPUT` command.

You can then use the `FormModifyObject(_SetFocus)` command to place the cursor back into the edit control; this forces the user to correct the entry.

Data entry forms that verify their fields usually allow a `CANCEL` button to override the verification. The program can check to see which button or object was selected by using `FormWait(_Peek)` function, which returns the ID of that object.

If the ID of the `CANCEL` button is returned (or the ID of some other object that precludes the need for verification), the program can just continue without checking the data.

`FormWait(_Peek)` returns 0 if the object that was selected and given the current focus does not notify the form.

`FormWait(_Peek)` only looks ahead at the object that was selected. The next regular `FormWait` will return the same ID as the `FormWait(_Peek)` function, unless the form is closed or the focus is explicitly set somewhere else in the form with the `FormModifyObject(_SetFocus)` command.

For an example of how you might use edit control verification, see *An Example of a Modal Form* later in this chapter.

Modal Pick and PickDrag

FormWait(_Pick) The `FormWait(_Pick)` function allows the user to select an object in a form without processing it.

`FormWait(_Pick)` returns the ID of the first object the user clicks on in the form. Objects that normally react to the click will not. Option buttons, for example, will not change state. Even objects that never notify the form, such as captions, will return their ID to a `FormWait(_Pick)`.

FormWait(_PickDrag) `FormWait(_Pick)` returns the ID of the first object the user clicks on in the form, regardless of whether or not that object normally notifies the form in response to a click (for example, captions). Objects such as option buttons and list boxes, that usually change state when clicked, will not. They will only return their ID number.

If an object has been grayed, it will not respond to a `_Pick` or a `_PickDrag`. This allows the program to control the items that the user can manipulate. Captions are gray when they are created, so they must be explicitly set to `_Normal` if they are to be used with `_Pick` or `_PickDrag`.

These modes are most useful when the program needs to manipulate the objects without regard for their actual use. A prototyping tool that interactively allows the user to create a form by positioning objects uses `FormWait(_Pick)` and `FormWait(_PickDrag)`.

A grid to which objects will automatically snap as they are moved can be added with the `FormSetGrid` command. For more information, see *Setting a Background Grid* later in this chapter.

An Example of a Modal Form

The following example creates a modal form that allows the user to indicate the number of shares of stock to either buy or sell, and whether a confirmation of the transaction should be sent to the client.

The processing is done with a `FormWait` loop, which waits for the OK or CANCEL button to be pressed. When the OK button is pressed, the values in the form (Shares, Transaction, Confirm) are queried with the `FormQStr` and `FormQNum` functions. If the number of shares is negative or zero, a warning is displayed and the user must correct the problem. This verification is not done if the CANCEL button is pressed instead.

```
' Manual\PG_15\STKFORM1.RLZ

'Create a new form and size it
MyForm = FormQUnique
FormNew(MyForm; "Stock Transaction Form", _Title)
FormControl(_Size; _Center, _Center, 40 pct, 60 pct)

'Put the objects in the form
FormSetObject(10, _GroupBox, "Transaction", \\  
    30 pct, 5 pct, 40 pct, 35 pct)
FormSetObject(20, _OptionButton, "Buy", 40 pct, \\  
    15 pct; 1)
FormSetObject(30, _OptionButton, "Sell", 40 pct, \\  
    25 pct)
FormSetObject(50, _CaptionLeft, \\  
    "Number of shares:", 5 pct, 51 pct)
FormSetObject(40, _TextBox, "", 60 pct, 50 pct, \\  
    30 pct, _Default)
FormSetObject(60, _CheckBox, "Send Confirmation", \\  
    _Center, 65 pct)
FormSetObject(1, _DefButton, "OK", _Left, _Bottom)
FormSetObject(2, _Button, "Cancel", _Right, _Bottom)
```



```

'Process the form and wait for a button to be
pressed
LOOP
    Selection = FormWait
    SELECT CASE Selection
    CASE 1      'OK / Enter
        Shares = StrToNum(FormQStr(40))
        IF Shares <= 0 THEN
            INPUT "Number of shares must be positive";
            EXIT SELECT
        END IF
        Transaction = FormQNum(20)
        Confirm = FormQNum(60)
        EXIT LOOP

    CASE 2      'Cancel / Escape
        Shares = 0
        EXIT LOOP
    END SELECT
END LOOP

'The form can be closed now
FormControl(_Close)

'If shares = 0 then the Cancel button was clicked
'Otherwise the transaction can be executed

IF Shares > 0 THEN
    INPUT Sprintf("P(0) shares were &.  \\
    Confirmation was &sent", shares, \\
    IF Transaction=1 THEN \\
    "bought" ELSE "sold", \\
    IF Confirm THEN "" ELSE "not");
ELSE
    INPUT "Transaction cancelled";
END IF

```

As an example of edit control notification, the check for a positive number of shares could have been done as soon as the user tried to leave the field. In such a case, the `_Notify` flag would have been set for the edit control with the `FormSetObject` command and the modal loop would have been as follows:

```
' Manual\PG_15\STKFORM2.RLZ
LOOP
  Selection = FormWait
  SELECT CASE Selection
  CASE 40 'Number of shares
    Shares = StrToNum(FormQStr(40))
    IF Shares <= 0 AND FormWait(_Peek) <> 2 THEN
      INPUT "Please respecify number of shares";
      FormModifyObject(40, _SetFocus)
    END IF

  CASE 1 'OK / Enter
    Shares = StrToNum(FormQStr(40))
    Transaction = FormQNum(20)
    Confirm = FormQNum(60)
    EXIT LOOP

  CASE 2 'Cancel / Escape
    Shares = 0
    EXIT LOOP
  END SELECT
END LOOP
```

Modeless Forms

When you design a form to operate with other forms and other application tools, it is called a modeless form. Unlike a modal form, a modeless form operates passively—it waits for the user to interact with it instead of demanding an immediate response. Other programs can run while a modeless form waits for input. The user can display and interact with several modeless forms or switch between a modeless form and the program window—the manner in which the user moves from window to window is unrestricted. Modeless forms also enable the user to conveniently operate a program through menus.

For example, a CA-Realizer program that displays a text file and lets the user edit it should be designed as a modeless form. In this way, the user can run other CA-Realizer programs or manipulate other CA-Realizer windows at the same time.

When you design a program with modeless forms, it expands the flexibility of the user interface. For example, the user can now choose from various modeless forms or bring up the Print Log to examine its contents.

Associating a Form Procedure

To create a modeless form, you associate a procedure with it using the `FormSetProc` command. Modeless forms automatically invoke their form procedure when CA-Realizer notifies them of an action.

The user can manipulate the passive controls in the form, such as option buttons and edit controls, without the program running. When the user activates an object (such as a button) that notifies the form, the form procedure will be invoked and processing can occur. After the form procedure is finished, the program stops again until the user activates another object. Modeless forms are notified of the same events as a modal form, but instead of explicitly asking which object was selected with a `FormWait`, the modeless form's form procedure is invoked.

Form Procedures

As with the other application tools, you can associate a procedure with a form. However, a form procedure is more discriminating in the way it works:

- User interaction with standard interface controls automatically invokes a form procedure. For example, a single click on a button or a double click in a list box triggers a form procedure.
- A user interacting with a tool inside a form does not invoke a form procedure. Application tools, whether they exist inside or outside a form window, only invoke their associated tool procedure.

The Structure of the Form Procedure

The single parameter that is passed to the form procedure is an array of information (called *params*) that contains the type of message and the object that was manipulated. For details about the values for each of the parameters, see "Overview of the CA-Realizer Programmable Application Tools."

The basic structure of a form procedure is as follows:

```
PROC MyFormProc(params)
  FormSelect(params[_FormNum])
  SELECT CASE params[_ItemNum]

    CASE 10
      SELECT CASE params[_Invoke]
        CASE _Click
          'Process object 10 click
        CASE _RightClick
          'Process object 10 right click
        'Process other possible messages
      END SELECT

    CASE 20
      'Process object 20

    Process other objects
  :
```

```

CASE 1
    'Process the OK button or ENTER key
CASE 2
    'Process the Cancel button or Escape key
CASE 0, -1
    'Process messages to the form itself
    'such as _Close

END SELECT
END PROC

```

The procedure outlined above is only an example. Not all forms have an OK and CANCEL button; therefore, a form could ignore objects 1 and 2.

Modeless Form Messages

The following table summarizes the possible messages that can be received by the form procedure. The `_Size`, `_MiddleClick`, `_RightClick`, and `_GetFocus` messages will not be received unless they are listed as modifiers to the `FormSetProc` command.

Message	Description
<code>_Change</code>	A <code>_ListBox</code> created with the <code>_Notify</code> style will generate a <code>_Change</code> message every time the selection is changed by the user.
<code>_Click</code>	The form (object 0) will be notified with a <code>_Click</code> message if the user left clicks the background of the form. Most objects in the form generate <code>_Click</code> messages by default, or when the <code>_Notify Style</code> flag is used when they are created with the <code>FormSetObject</code> command.
<code>_Custom</code>	Can only be generated by a custom control.
<code>_CustomNow</code>	Can only be generated by a custom control.

Continued

Continued

Message	Description
<code>_DragNDrop</code>	The form that received the drop will receive a <code>_DragNDrop</code> message.
<code>_GetFocus</code>	The objects in the form will receive <code>_GetFocus</code> messages when the user selects them or tabs to them. The form itself (object 0) will generate a <code>_GetFocus</code> message when the form is made the topmost. This message is not generated unless it was specified with the <code>FormSetProc</code> command.
<code>_MiddleClick</code>	The objects in the form that respond to left button clicks and the form itself (object 0) will generate <code>_MiddleClick</code> messages when the middle mouse button is clicked. This message is not generated unless it was specified with the <code>FormSetProc</code> command.
<code>_RightClick</code>	The objects in the form that respond to left button clicks and the form itself (object 0) will generate <code>_RightClick</code> messages when the right mouse button is clicked. This message is not generated unless it was specified with the <code>FormSetProc</code> command.
<code>_Size</code>	The form will be notified with a <code>_Size</code> message to object 0 whenever the form is resized by the user. This message is not generated unless it was specified with the <code>FormSetProc</code> command.

Edit Control Verification

Edit control verification was discussed earlier in the context of modal forms. You can use a similar method with modeless forms. When an edit control is left because the mouse is clicked in another object or because the user pressed the TAB key, the form procedure will be invoked and `_ItemNum` will be the ID of the edit control.

If another notifying object (such as a button) is selected after the form procedure has finished processing the edit control, it will be invoked again immediately for the next object. The program can identify the next object by using the `FormWait(_Peek)` function, which returns the object's number, or 0 if the next object does not notify the form.

If you want the program to force the edit control to retain the current focus, you can use `FormModifyObject(_SetFocus)`. Controlling the focus with this command prevents the clicked-on object from notifying the form. In other words, `_SetFocus` cancels any pending notification.

Modeless Pick and PickDrag

The entire modeless form can be placed into a mode where objects are selectable or draggable with the `FormControl(_SetMode)` command:

```
FormControl(_SetMode; FormMode)
```

The valid values for *FormMode* are `_Pick`, `_PickDrag`, or `_Normal`.

When the form is in `_Pick` or `_PickDrag` mode, the objects will no longer react as they normally do. Instead, they will notify the form with a `_Click` message when they are clicked (in `_Pick` mode) or dragged (in `_PickDrag` mode).

Refer to the earlier discussion of `_Pick` and `_PickDrag` for more information.

An Example of a Modeless Form

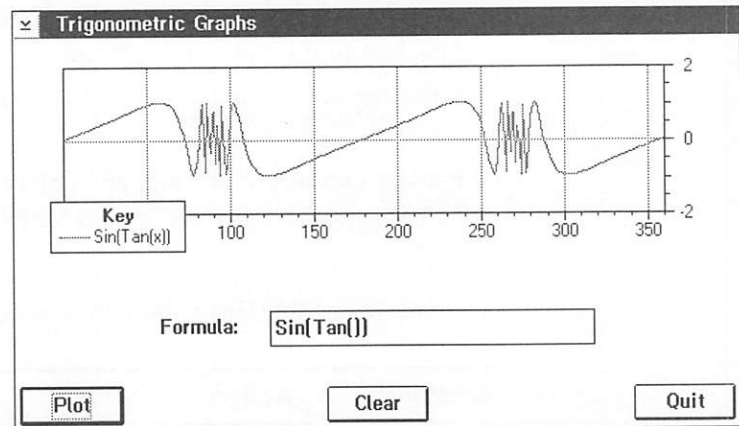
The following program graphs trigonometric functions. The form contains a chart and an edit control for entering formulae. The formula for the function to be plotted should contain empty parentheses where the variable belongs, as in `Sin()` or `Tan(Tan())`. Graphs are drawn from 1 to 360 degrees, each in a different color, and a chart key is maintained.

```
' Manual\PG_15\TRIGPLOT.RLZ

PROC formprocTrig(params)
  FormSelect(params[_FormNum])
  SELECT CASE params[_ItemNum]
  CASE 1 'Plot
    Formula = SubStr$(FormQStr(30), "()", "(x)")
    x = Index(360) * (3.1416/180)
    EXECUTE "Results = " + Formula
    PlotColor = PlotColor + 1
    ChartSetColor(PlotColor)
    ChartLine(Results)
    ChartSetKey(Formula)
  CASE 40 'Clear
    ChartControlPane(_Clear)
    ChartControlKey(_Clear)
    PlotColor = _Red
  CASE 50 'Quit
    FormControl(_Close)
  END SELECT
END PROC

formTrig = FormQUnique
FormNew(formTrig; "Trigonometric Graphs", _Title)
FormControl(_Size; _Center, _Center, 80 pct, 80 pct)
FormSetObject(10, _Chart, "", 5 pct, 5 pct, \
  90 pct, 50 pct)
FormSetObject(20, _CaptionLeft, "Formula: ", \
  20 pct, 71 pct)
FormSetObject(30, _TextBox, "Sin(Tan())", 35 pct, \
  70 pct, 45 pct, _Default)
FormSetObject(1, _DefButton, "Plot", _Left, _Bottom)
FormSetObject(40, _Button, "Clear", _Center, \
  _Bottom)
FormSetObject(50, _Button, "Quit", _Right, _Bottom)
FormSetProc(formprocTrig)
PlotColor = _Red
FormControl(_Show)
ChartControlKey(_Show)
```


When you choose the Plot button, the following chart is displayed:



The formula is extracted from the edit control and converted to a CA-Realizer command by substituting (*x*) for the empty parentheses and prefixing the formula with Results =. This command is then run with the EXECUTE command, and the results are put in the chart. All of this happens when the user presses the PLOT button or the ENTER key. (ENTER is synonymous with PLOT in this case because PLOT is a default button and has been assigned an ID of 1.)

Tip: Other interesting plots to run are: Tan(Tan()), Sin(Cos()), and Cos(Sin()).

Setting a Background Grid

The `FormSetGrid` command adds, removes, or alters a grid in the background of the current form:

```
FormSetGrid(GridMode [[, Left, Top], Width, Height]  
            [, Color])
```

When a grid is active, the top left of all objects that you drag within the form are automatically snapped to the nearest grid lines.

The following table lists valid values for *GridMode*:

GridMode	Action
<code>_Off</code>	Turns the grid off.
<code>_Dot</code>	Displays the grid as dots.
<code>_Line</code>	Displays the grid as lines.
<code>_Hidden</code>	Hides the grid from view. However, objects still snap to the grid.

If you specify *Left* and *Top* (in pixels), CA-Realizer adjusts the grid to intersect that point. If not, the grid has its origin at the upper-left corner of the form. If you specify *Width* and *Height*, the width and height of the grid is set to these pixel values. Setting either of these values to 0 removes the grid. *Width* removes vertical lines and *Height* removes the horizontal grid.

A form can be placed in `_Pick` or `_PickDrag` mode with one of the following:

```
ID = FormWait(_Pick)  
ID = FormWait(_PickDrag)  
FormControl(_SetMode; _Pick)  
FormControl(_SetMode; _PickDrag)
```

Run the following example to see how `_SetMode` is used with the `_PickDrag` modifier:

```
FormNew( ; "A Form with Grids")
FormSetColor(_Blue; _Background)
FormSetGrid(_Line, 20, 30; _Red)
FormSetObject(10, _Button, "Move Me", \
    _Center, _Center)
FormControl(_SetMode; _PickDrag)
FormControl(_Show)
```

Adding Accelerator Keys

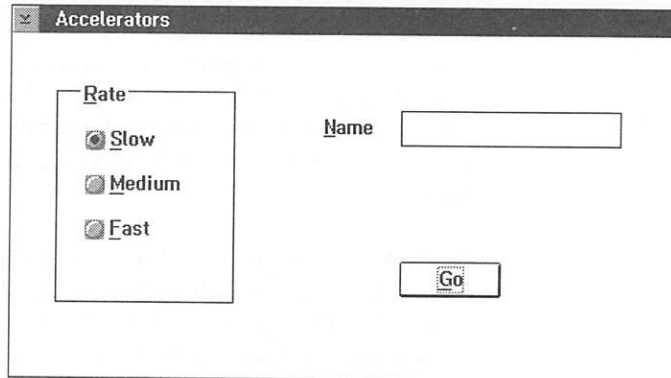
Accelerator keys are keystroke combinations that allow the user to activate a form object, such as a button, without the mouse.

Accelerators can be added when an object is first created with the `FormSetObject` command. To add accelerators to a form object when it is created, simply place an ampersand (&) before one of the letters in the object's *Text*. This letter will be underlined and, when combined with the ALT key, will be functionally equivalent to clicking the object with the left mouse button.

When an accelerator is used to generate a `_Click` message, the *params[_XPos]* and *params[_YPos]* values will be set to 32,767. To include a literal ampersand (&) in the *Text* of an object, place two ampersands (&&) in the position where you want a single ampersand to display.

The following example places accelerators on a number of objects:

```
' Manual\PG_15\ACCEL.RLZ
FormNew(; "Accelerators", _Title)
FormControl(_Size; _Center, _Center, 450, 250)
FormSetObject(10, _GroupBox, "&Rate", 30, 30, \
120, 150)
FormSetObject(20, _OptionButton, "&Slow", 50, 60; \
1, 1)
FormSetObject(30, _OptionButton, "&Medium", 50, \
90; 0, 1)
FormSetObject(40, _OptionButton, "&Fast", 50, \
120; 0, 1)
FormSetObject(50, _Button, "&Go", 260, 150)
FormSetObject(60, _CaptionLeft, "&Name", 200, 54; \
_Notify)
FormSetObject(70, _TextBox, "&Text<Box", 260, 50, \
150, _Default)
FormModifyObject(70, _Clear)
FormControl(_Show)
```



Note that the group box title and the option buttons contained within it each have the accelerators, R, S, M and F, respectively. In addition, the Name edit control and Go button have the accelerators, N and G, respectively.

Edit controls and list boxes can also have accelerators, even though their text is not displayed. The accelerator will still give the focus to the object. In the above example, ALT+T will give the focus to the edit control.

Captions and group boxes can have accelerators, but in most cases there is nothing for them to do. In the preceding form example, it would be useful for the ALT+N combination attached to the NAME caption to notify the edit control and give it focus. Likewise, the ALT+R combination for the RATE group box could give the focus to one of the radio buttons. This effect can be achieved by using the `_AccelNotify` modifier when the caption or group box is created.

Drag and Drop

Another powerful feature that is supported by CA-Realizer forms is *drag and drop*. With drag and drop, there are two types of objects, usually represented by bitmap icons. Some icons are set as draggable objects, and others are set as drop receivers.

When draggable objects are clicked and dragged, the mouse cursor changes to a circle. When the cursor is moved over a drop receiver, the dragged object highlights to indicate a valid action. If the user releases the mouse button, the form procedure is notified with all the information about the drag and drop.

Forms themselves can also be drop receivers. This allows for the creation of programs that move icons from one form to another.

Setting Draggable Objects and Drop Receivers

To set objects as draggable objects or drop receivers, use the `FormModifyObject` command as follows:

```
FormModifyObject(ID [, Attribute [, Text]
                [, Font] [, Left, Top [, Width,
                Height]]] [; Mod1 [... , ModN]])
```

The following table lists the valid values for *Attribute*:

<i>Attribute</i>	Description
<code>_Drag</code>	Sets the object as draggable.
<code>_DragOff</code>	Sets the object as non-draggable.
<code>_Drop</code>	Sets the object as a drop receiver.
<code>_DropOff</code>	Sets the object as <i>not</i> a drop receiver.

To make an entire form a drop receiver, use the `FormControl(_SetDrop)` command as follows:

```
FormControl(_SetDrop ; Drop)
```

If *Drop* is 1, the form accepts drag and drop objects onto the form background. When this occurs, the form procedure is notified with *params[_Invoke]* set to `_DragNDDrop` and *params[_ItemNum]* set to 0. If *Drop* is 0, the form does not accept drag and drop objects.

Processing Drag and Drop Messages

When a draggable object is released over a droppable object or form, a `_DragNDrop` message is sent to the form procedure for the receiver. The parameter array elements are described below:

Element	Description
<code>params[_DragForm]</code>	The form from which the object was dragged.
<code>params[_DragItem]</code>	The ID of the object that was dragged.
<code>params[_FormNum]</code>	Form number of the receiver.
<code>params[_Invoke]</code>	Set to <code>_DragNDrop</code> .
<code>params[_ItemNum]</code>	Item number of the receiver, or 0 if the form is the receiver.
<code>params[_XPos]</code> and <code>params[_YPos]</code>	The position in the receiver where the object was dropped.

A Drag and Drop Example

The following example creates a form with 25 objects. All of them are made draggable, except the middle one, which is set as a drop receiver. If an object is dragged onto the middle icon, the dragged object is removed from the form.

```
' Manual\PG_15\DRAGDROP.RLZ

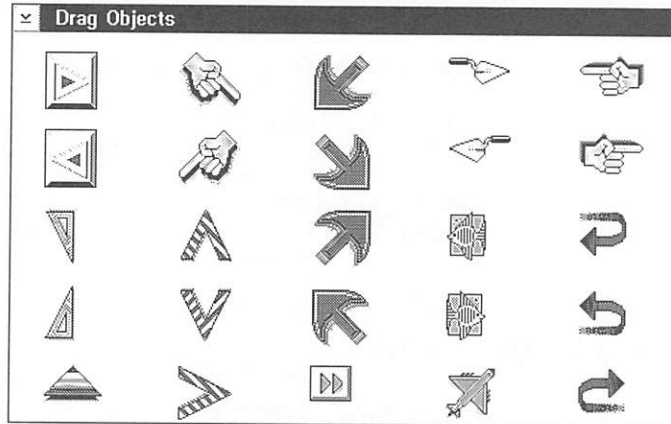
PROC formproc(params)
  FormSelect(params[_FormNum])
  SELECT CASE params[_Invoke]
    CASE _DragNDrop
      'Destroy the dragged item
      item = params[_DragItem]
      FormModifyObject(item, _Close)
    END SELECT
  END PROC
```

```

FormNew(1; "Drag Objects", _Title)
FormControl(_Size; _Center, _Center, 70 pct, 70 pct)
FOR i = 1 to 5
  FOR k = 1 to 5
    n = i*10 + k
    FormSetObject(n, _Bitmap, \\
      Sprint("CLIPART_\\ARROW(0).BMP", n), \\
      i * 20 -15 pct, k * 20 - 15 pct)
    'Make the object draggable
    FormModifyObject(n, _Drag)
  NEXT k
NEXT i
FormSetProc(formproc)
FormControl(_Show)

'Set the middle object as the drop receiver
FormModifyObject(33, _Drop)
FormModifyObject(33, _DragOff)

```



Chapter 16

Graphics Tablets

In This Chapter	16-1
Creating Images	16-1
Creating a Graphics Tablet	16-2
Selecting a Tablet	16-3
Drawing on a Tablet	16-4
Pens and Brushes	16-5
Drawing Lines	16-7
TabletLine	16-7
TabletLineTo	16-8
TabletMoveTo	16-8
TabletPolyline	16-8
Drawing Simple Shapes	16-9
Rectangles	16-9
Rounded Rectangles	16-9
Ellipses	16-10
Arcs	16-10
Chords	16-10
Pies	16-11
Example	16-11
Drawing Polygons	16-12
Drawing Bitmaps and Individual Pixels	16-13
Returning the Color of an Individual Pixel	16-14
Drawing Text	16-15
Trapping Mouse Clicks in a Tablet	16-16

Buffering Modes	16-17
Metafile Mode	16-17
BufferedMetafile Mode	16-17
Bitmap Mode	16-17
Controlling the Redraw	16-18
Undoing Commands	16-20
A Tablet in Action—Tic Tac Toe	16-21
A Tablet in Action—Simple Draw!	16-23

Chapter 16

Graphics Tablets

In This Chapter

A graphics tablet is a region in a form in which lines, rectangles, polygons, ellipses, and text are drawn. Graphics tablets are useful for creating drawings, as well as for generating plots that either do not need, or fit within, the overall structure of CA-Realizer charts. In this chapter, you will learn about the various CA-Realizer commands for creating graphics, as well as how to control their content and format.

Creating Images

The graphics tablet can be a powerful canvas for your ideas. Its power and flexibility allow you to create images in both *vector* and *bitmap* formats.

Vector

Vector graphics are based upon the manipulation of relatively simple mathematical constructs—points and lines. You commonly use vector graphics to generate drawings. CA-Realizer provides many drawing commands that can be used for both simple and complex shapes. For example, an arc is a mathematical construct that you can generate using the `TabletArc` command of CA-Realizer.

Bitmap

On the other hand, bitmap graphics have more in common with freehand painting—they operate by switching individual pixels off and on. CA-Realizer commands enable you to query and manipulate individual pixels—for instance, you can change the color of a pixel by using the `TabletSetPixel` command.

Creating a Graphics Tablet

Recall that the graphics tablet tool can exist only inside a form. Graphics tablets can be placed in a form with any of the standard CA-Realizer form objects, or with other programmable application tools. You can place multiple tablets in the same form, or a single tablet can occupy the entire form.

Like other tools in a form, you create tablets with the `FormSetObject` command:

```
FormSetObject(ObjectNum, _Tablet, "", [Font,]  
             Left, Top, [, Width, Height] [, Style])
```

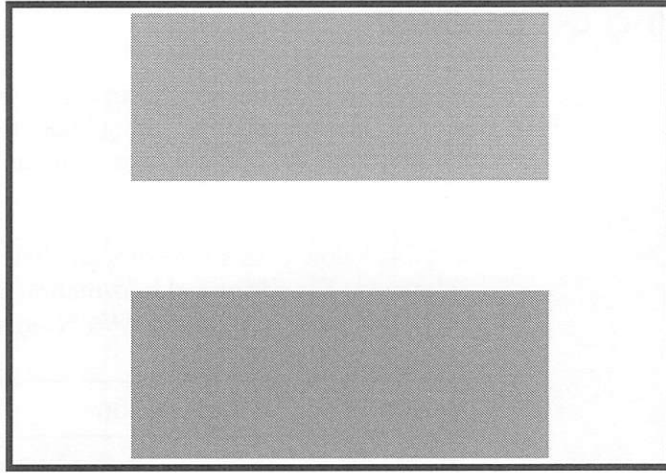
As always, you can use CA-Realizer position notation to specify the size and position of the tablet within the form. Coordinates within the tablet, however, are always specified in pixels.

Note: You will want to exercise care when developing a tablet application on one system resolution—for example, using VGA and delivering it on Super VGA.

If you specify *Font*, CA-Realizer uses it when text is drawn with the `TabletText` command. If you specify `_FillForm` as the value for the optional *Style* modifier, the tablet resizes itself to the full size of the form whenever the user resizes the form.

The following program fragment creates a form and two tablets within it. The tablets are 300 pixels wide and 120 pixels high. The pixels are numbered from 0 to 299, with (0, 0) as the upper-left corner of the tablet.

```
'Create a form with two tablets  
FormNew(10;"Graphics Tablet")  
FormControl(_Size; _Center, _Center, 60 pct, 60 pct)  
FormSetColor(_LightGray; _Field)  
FormSetObject(20, _Tablet, "", _Center, _Top, \  
              300 pxl, 120 pxl)  
FormSetColor(_Cyan; _Field)  
FormSetObject(30, _Tablet, "", _Center, _Bottom, \  
              300 pxl, 120 pxl)  
FormControl(_Show)
```



Selecting a Tablet

Since you can incorporate several tablets in a program, you must first select a tablet before you can draw in it.

All tablet commands affect the current tablet. A tablet becomes current when it is created, or if it is selected with the `TabletSelect` command. The `TabletSelect` command works like the other `TOOLSelect` commands, such as `ChartSelect` or `FormSelect`:

```
TabletSelect (TabletNum [, FormNum])
```

When a tablet is selected with the `TabletSelect` command, all subsequent tablet commands apply to this tablet. *TabletNum* is the number of the tablet, as specified in the `FormSetObject` command. If the tablet is not within the current form, specify *FormNum* as well.

Drawing on a Tablet

CA-Realizer provides a range of commands that you can use for drawing lines and shapes, filling lines and shapes with color, manipulating bitmaps and pixels, and annotating drawings with text.

The table below presents a complete list of the tablet drawing commands. For additional information about each of these commands, refer to the *CA-Realizer Language Reference Guide*.

Command	Description
TabletArc	Draws an elliptical arc.
TabletBitmap	Inserts a bitmap into a tablet.
TabletChord	Draws a chord.
TabletEllipse	Draws an ellipse.
TabletLine	Draws a line between two points.
TabletLineTo	Draws a line from the current position to a specified point.
TabletPie	Draws a pie-shaped wedge.
TabletPolygon	Draws a closed polygon from a list of specified vertices.
TabletPolyLine	Draws a series of lines between a set of given points.
TabletPolyPolygon	Draws two or more polygons.
TabletRectangle	Draws a rectangle.
TabletRoundRect	Draws a rectangle with rounded corners.
TabletText	Draws text in the tablet.

Pens and Brushes

Objects can be drawn using a specified *pen* and *brush*. A pen is used to draw the frame of an object in a specified color and thickness. Brushes are used to set the “fill color” of an object.

Use the `TabletPen` and `TabletBrush` commands to set the characteristics of your pens and brushes.

TabletPen

`TabletPen` sets the color and width of the pen in the current tablet.

```
TabletPen(Color [, Width])
```

Color is one of the predefined CA-Realizer colors or a three-element array of RGB values ranging from 0 to 1. *Width* specifies the width of the pen in pixels.

The default is a 1-pixel black pen.

TabletBrush

`TabletBrush` sets the color of the brush in the current tablet:

```
TabletBrush(Color)
```

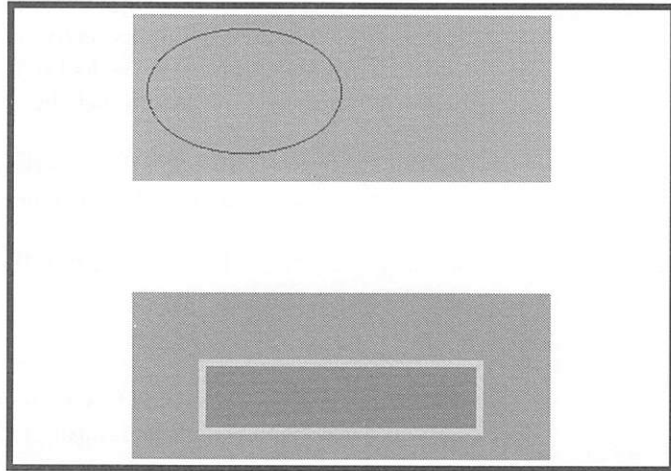
Color is one of the predefined CA-Realizer colors or a three-element array of RGB values ranging from 0 to 1. The default is `_Empty` (that is, transparent or unfilled).

You can add some simple shapes by adding the following lines to the program created earlier in this chapter:

```
'Draw the outline of an ellipse in blue
TabletSelect(20)
TabletPen(_Blue)
TabletEllipse(10, 10, 150, 100)
```

```
'Draw a green square with a thick yellow outline
TabletSelect(30)
TabletPen(_Yellow; 5)
TabletBrush(_Green)
TabletRectangle(50, 50, 250, 100)
```

Running this program displays the following tablet:



The new program draws an ellipse and a rectangle in the two tablets that were already defined to the form (identified by the object numbers of 20 and 30).

Both the `TabletEllipse` and `TabletRectangle` commands take four parameters. In each command, the first two parameters indicate the *X* and *Y* values in pixels, relative to the upper-left corner of the tablet, at which to start drawing the upper-left corner of each form object, while the last two represent the *X* and *Y* values for their bottom-right corner.

Notice the difference between this method and the one that other CA-Realizer tools use. The other tools normally use the width and the height. Also note that the units for these coordinates are not specified, because all tablet coordinates must be given in pixels.

Drawing Lines

Lines are one of the most fundamental of the drawing tools, and CA-Realizer provides several commands for creating them.

TabletLine

TabletLine draws a line in the current tablet.

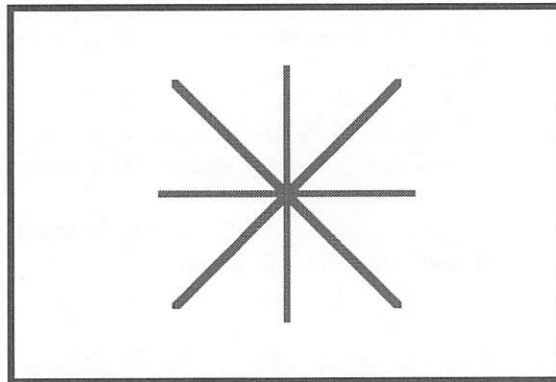
```
TabletLine(X1, Y1, X2, Y2)
```

The line is drawn from the point (X1, Y1) to the point (X2, Y2). All coordinates must be given in pixels.

Try this example using the TabletLine command:

```
FormNew
FormSetObject(10, _Tablet, "", _Center, _Center, \
    200 pxl, 200 pxl)
FormControl(_Show)

'Draw a large asterisk
TabletPen(_Red; 5)
TabletLine(20, 20, 180, 180)
TabletLine(20, 180, 180, 20)
TabletLine(100, 10, 100, 190)
TabletLine(10, 100, 190, 100)
```



TabletLineTo

TabletLineTo draws a line from the current position to the point specified by (X, Y).

```
TabletLineTo(X, Y)
```

X and Y must be given in pixels.

TabletMoveTo

You can use the TabletMoveTo command to set the current position. The current position is also updated with each TabletLineTo command.

```
TabletMoveTo(X, Y)
```

TabletPolyline

TabletPolyline draws a set of lines between each consecutive pair of points in (XPoints, YPoints).

```
TabletPolyline(XPoints, YPoints [, BooleanArray])
```

If you specify *BooleanArray*, CA-Realizer only draws lines from points in which the corresponding element in *BooleanArray* is non-zero.

XPoints and *YPoints* are arrays that must contain the same number of elements. Specify the coordinates in pixels.

For example, the asterisk example above could have been drawn with the following:

```
x = {20, 180, 20, 180, 100, 100, 10, 190}  
y = {20, 180, 180, 20, 10, 190, 100, 100}  
b = {1, 0, 1, 0, 1, 0, 1, 0}  
TabletPolyline(x, y, b)
```

Drawing Simple Shapes

While sufficient for drawing many shapes, the line drawing commands alone are not always the best solution for constructing shapes. For example, in order to generate an arc, you have to loop through the `TabletLineTo` command or the `TabletSetPixel` command.

A much faster, and simpler, alternative is to use the following commands to draw the geometric shapes.

Rectangles

`TabletRectangle` draws a rectangle in the current tablet.

```
TabletRectangle(Left, Top, Right, Bottom)
```

The upper-left corner of the rectangle is located at (*Left*, *Top*) and the lower-right corner is located at (*Right*, *Bottom*).

Rounded Rectangles

`TabletRoundRect` draws a rectangle with rounded corners in the current tablet.

```
TabletRoundRect(Left, Top, Right, Bottom,  
                RWidth, RHeight)
```

RWidth and *RHeight* are the width and height, respectively, of the ellipse used to draw the rounded corners.

Ellipses

TabletEllipse draws an ellipse in the current tablet.

`TabletEllipse(Left, Top, Right, Bottom)`

The ellipse is drawn within the “bounding” rectangle defined by (*Left*, *Top*) and (*Right*, *Bottom*).

Note: You can also draw a circle with radius r centered at (cx , cy) with `TabletEllipse( $cx-r$ ,  $cy-r$ ,  $cx+r$ ,  $cy+r$ )`.

Arcs

TabletArc draws an elliptical arc.

`TabletArc(Left, Top, Right, Bottom,
 XStart, YStart, XEnd, YEnd)`

XStart and *YStart* are the coordinates of the arc’s starting point; *XEnd* and *YEnd* are the coordinates of the arc’s ending point.

Chords

TabletChord draws a chord, which is a closed figure bounded by the intersection of an ellipse and a line segment.

`TabletChord(Left, Top, Right, Bottom,
 XStart, YStart, XEnd, YEnd)`

In this case, *XStart* and *YStart* are the coordinates of the line segment’s starting point. *XEnd* and *YEnd* are the coordinates of the line segment’s ending point.

Pies

`TabletPie` draws a pie-shaped wedge by drawing an elliptical arc whose center and two endpoints are joined by lines.

```
TabletPie(Left, Top, Right, Bottom,
          XStart, YStart, XEnd, YEnd)
```

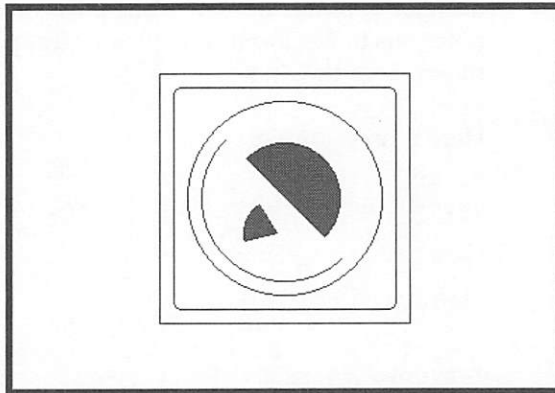
XStart and *YStart* are the coordinates of the arc's starting point; *XEnd* and *YEnd* are the coordinates of the arc's ending point.

Example

The following example demonstrates these shapes:

```
FormNew
FormSetObject(10, _Tablet, "", _Center, _Center, \
              200 pxl, 200 pxl)
FormControl(_Show)

TabletRectangle(10, 10, 190, 190)
TabletRoundRect(20, 20, 180, 180, 10, 15)
TabletEllipse(30, 30, 170, 170)
TabletArc(40, 40, 160, 160, 50, 150, 150)
TabletBrush(_Red)
TabletChord(60, 60, 140, 140, 130, 130, 70, 70)
TabletPie(70, 100, 120, 150, 85, 110, 90, 125)
```



The outline of these shapes is always drawn with the current pen color and width. CA-Realizer always fills the interior with the current brush color, if any.

Drawing Polygons

The previous commands have all taken some form of individual *X* and *Y* coordinates as their arguments. The following commands allow you to draw complex shapes by providing the functions with *arrays* of points instead.

TabletPolygon

TabletPolygon draws a polygon by first drawing a line between each consecutive pair of points (*XPoints*, *YPoints*), and then a line from the last point back to the first.

```
TabletPolygon(XPoints, YPoints)
```

XPoints and *YPoints* are arrays that must have the same number of elements.

TabletPolyPolygon

TabletPolyPolygon draws two or more disjoint or overlapping polygons.

```
TabletPolyPolygon(XPoints, YPoints, NumPoints)
```

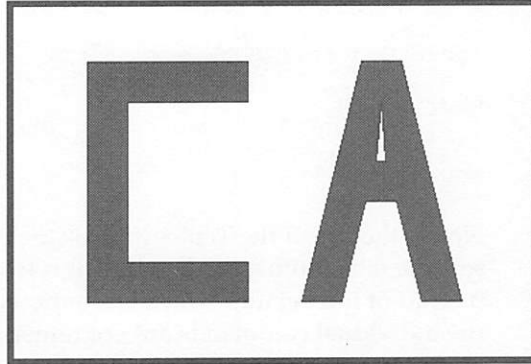
XPoints and *YPoints* are arrays that must have the same number of elements.

NumPoints is an array of numerics, each of which specifies the number of points in each of the polygons. If you want the polygons to be closed, specify an extra point at the end that is the same as the first.

Here is an example:

```
FormNew
FormSetObject(10, _Tablet, "", _Center, _Center, \\  
              300 pxl, 200 pxl)  
FormControl(_Show)  
  
TabletBrush(_Blue)  
x = {10, 130, 130, 40, 40, 130, 130, 10}  
y = {10, 10, 40, 40, 160, 160, 190, 190}  
TabletPolygon(x, y)
```

```
x = {170, 200, 218.75, 241.25, 260, 290, 245, \\  
    215, 170, 226.25, 233.75, 230, 226.25}  
y = {190, 190, 115, 115, 190, 190, 10, 10, 190, \\  
    85, 85, 40, 85}  
TabletPolyPolygon(x, y, {9, 4})
```



Drawing Bitmaps and Individual Pixels

CA-Realizer also permits you to incorporate bitmaps into your tablets, as well as to manipulate individual pixels.

TabletBitmap

You can include a bitmap in a drawing by using the `TabletBitmap` command:

```
TabletBitmap(Bitmap, Left, Top, Width, Height)
```

Bitmap is a bitmap name or a picture number. *Left* and *Top* are the coordinates at which to place the upper-left corner of the bitmap. *Width* and *Height* represent the width and height at which to draw the bitmap.

TabletSetPixel

You can paint an image on the graphics tablet using the `TabletSetPixel` command:

```
TabletSetPixel(X, Y, Color)
```

`TabletSetPixel` sets the pixel at the specified coordinates (*X*, *Y*) using the given color (*Color*). *Color* is one of the predefined colors or an array of RGB values:

The following program creates a tablet and proceeds to fill it with randomly colored pixels at random locations:

```
FormNew
FormSetObject(10, _Tablet, "", _Center, _Center, \\  
              200 pxl, 200 pxl)
FormControl(_Show)
TabletControl(_SetMode; _Bitmap)

LOOP
  TabletSetPixel(Rnd * 200, Rnd * 200, \\  
                Ceil(Rnd * 7))
END LOOP
```

Notice the use of the `TabletControl(_SetMode)` command. This sets the tablet into a mode where it is treated as a bitmap, instead of the default, which is a buffered metafile. As a result, the individual commands are not remembered and cannot be undone. Bitmap mode is faster and should always be used when a very large number of objects will be added to the form. Refer to *Buffering Modes* later in this chapter for more information.

Returning the Color of an Individual Pixel

Painting programs work by changing the color of individual pixels. This is somewhat analogous to switching individual pixels on and off.

To query the present color of a pixel, use the `TabletGetPixel` command:

```
Color = TabletGetPixel(X, Y)
```

This command retrieves the color of the pixel at the coordinates specified by (X, Y). Depending on the number of colors that your screen can handle, the color of a pixel inside a filled region may not be equal to the color that was used to fill it. For example, an orange region is sometimes made up of alternating red and yellow pixels (a *dither* pattern).

Drawing Text

Use the `TabletText` command and `TabletTextColor` command when you want to mix both text and graphics in a tablet.

`TabletText`

`TabletText` draws the text, *Text*, in the current tablet, using the current text color and the font specified when the tablet was created with the `FormSetObject` command.

```
TabletText(Text, Left, Top [, Right, Bottom]  
          [, Justify])
```

CA-Realizer places the text at the upper-left corner (*Left*, *Top*). If you specify *Right* and *Bottom*, CA-Realizer clips the text within the bounding rectangle defined by (*Left*, *Top*) and (*Right*, *Bottom*).

You can justify the text within the bounding rectangle by setting *Justify* to `_Left`, `_Center` or `_Right`. If you use *Justify*, define the bounding rectangle by specifying *Right* and *Bottom*.

By default, CA-Realizer left-justifies the text.

`TabletTextColor`

`TabletTextColor` sets the color that you use to draw text with the `TabletText` command.

```
TabletTextColor(Color)  
TabletTextColor(Red, Green, Blue)
```

Color should be one of the predefined CA-Realizer colors or a three-element array of RGB values ranging from 0 to 1.

The default text color is `_Black`.

Trapping Mouse Clicks in a Tablet

Interactive programs using the tablet might want to process mouse clicks inside it. When the user clicks inside the tablet, CA-Realizer notifies the form containing the tablet. This causes a `FormWait` to return or it invokes a form procedure to respond to the click.

The tablet remembers the coordinates of the last place the user clicked. You can determine this point using the `TabletMouseX` and `TabletMouseY` commands:

```
XLocation = TabletMouseX  
YLocation = TabletMouseY
```

These commands return the X and Y coordinates of the last point in the tablet that the user clicked. CA-Realizer returns the value in pixels and only guarantees its accuracy immediately after the tablet notifies the form procedure or after a `FormWait` returns.

Try the following program. It draws a red dot wherever the mouse is clicked:

```
PROC FormProc(params)  
  SELECT CASE params[_ItemNum]  
  CASE 10 'Tablet  
    x = TabletMouseX  
    y = TabletMouseY  
    TabletBrush(_Red)  
    TabletEllipse(x-5, y-5, x+5, y+5)  
    TabletControl(_Show)  
  END SELECT  
END PROC  
  
FormNew  
FormSetObject(10, _Tablet, "", _Center, _Center, \\  
  100 pct, 100 pct)  
FormControl(_Show)  
TabletControl(_SetMode; _Bitmap)  
FormSetProc(FormProc)
```

The sample drawing programs at the end of this chapter also provide examples of tablets that accept mouse clicks.

Buffering Modes

A graphics tablet can be set to one of three modes that determine what it remembers about the commands that have been drawn and how it buffers the picture. The mode can be changed with the `TabletControl(_SetMode)` command.

Metafile Mode

In `_Metafile` mode, the tablet remembers each of the commands that were used, and replays them every time the tablet needs to be drawn. This allows graphic elements to be removed and reveal what is beneath them.

The drawback to this method is the speed of redrawing, the extra memory required, and the limit on the number of graphic elements that can be placed in a tablet.

BufferedMetafile Mode

The `_BufferedMetafile` mode is similar to `_Metafile` mode, except that a bitmap version of the metafile is kept in memory to speed up the redraw speed. The only time the commands are replayed, is when an object is removed with the `TabletUndo` command.

Note: This is the default mode when the tablet is created.

Bitmap Mode

If the tablet does not need to utilize the undo feature, it should be set to `_Bitmap` mode. In this mode, commands are drawn straight to a bitmap and then forgotten. There is no additional memory overhead for each command, and, therefore, no limit to the number of commands that can be used.

Controlling the Redraw

When a number of tablet commands are used to create a display, the intermediate results are shown to the user since the tablet is redrawn for each command.

A tablet's automatic redraw can be turned off with the `TabletControl(_SetRedraw; 0)` command. In this case, the tablet will not be shown until an explicit `TabletControl(_Show)` is used. The redraw can be turned back on with the `TabletControl(_SetRedraw; 1)` command.

The following program, which illustrates the visual differences when redraw is on and off, draws a globe using a number of different sized ellipses when the Redraw button is pressed. If the check box is on, the redraw is set to 1 and each command is shown. If the check box is off, the picture will not be displayed until it has been completely redrawn.

```
' Manual\PG_16\GLOBE.RLZ

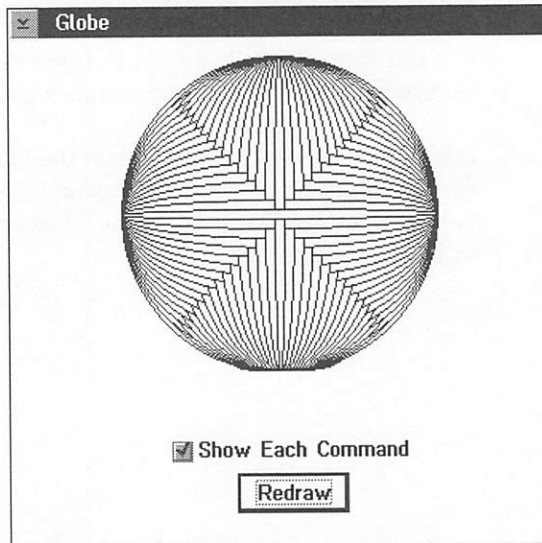
PROC DrawGlobe(redraw)
  GrowthRate = 6
  d1 = 0
  d2 = 200
  TabletControl(_Clear)
  TabletControl(_Show)
  TabletControl(_SetRedraw; redraw)
  WHILE d1 < 100
    TabletEllipse(d1, 0, d2, 200)
    TabletEllipse(0, d1, 200, d2)
    d1 = d1 + GrowthRate
    d2 = d2 - GrowthRate
  END WHILE
  TabletControl(_SetRedraw; 1)
  TabletControl(_Show)
END PROC
```

```

PROC formproc(params)
  FormSelect(params[_FormNum])
  SELECT CASE params[_Invoke]
  CASE _Click
    SELECT CASE params[_ItemNum]
    CASE 1
      DrawGlobe(FormQNum(30))
    END SELECT
  END SELECT
END PROC

FormNew(FormQUnique; "Globe", _Title+_Close)
FormControl(_Size; _Center, _Top, 60 pct, 70 pct)
FormSetObject(20, _Tablet, "", _Center, 5 pct, \
  200 pxl, 200 pxl)
FormSetObject(30, _CheckBox, "Show Each Command", \
  _Center, 82 pct)
FormSetObject(1, _DefButton, "Redraw", _Center, \
  _Bottom)
FormSetProc(formproc)
FormControl(_Show)
TabletControl(_SetMode; _Bitmap)
DrawGlobe(0)

```



Undoing Commands

When used in `_Metafile` or `_BufferedMetafile` mode, the tablet maintains a list of the objects that were drawn inside it. When it is updated, the tablet erases its drawing region and redraws the objects in the order they were added.

By default, `TabletControl(_Show)` redraws all the objects. If the tablet contains many ellipses or polygons, this update might take a few seconds. If you add a new object and want it to be shown without forcing the rest of the tablet to be redrawn unnecessarily, you can specify the number of objects to draw as a modifier to `TabletControl(_Show)`.

TabletUndo

Commands can be removed from the end of this list with the `TabletUndo` command.

`TabletUndo(NumObjects)`

The last *NumObjects* commands are removed. If *NumObjects* is not specified, only the last command is undone.

`TabletUndo` cannot be used when the tablet is in `_Bitmap` mode. The equivalent is to explicitly *undraw* the object (that is, draw over it in the background color). However, objects underneath will not be redrawn.

A Tablet in Action—Tic Tac Toe

Here is a larger program that uses the tablet as a playing board for a game of Tic Tac Toe. Most of the game logic has been left out to emphasize the use of the tablet. The game does not play against a person, nor does it determine the winner of a game. It does, however, draw the board and draws the pieces when the mouse is clicked in a square. Think of it as a computerized chalk board.

```

*****
* Tic Tac Toe Board Program *
*****

' Manual\PG_16\TICTACTO.RLZ

PROC DrawBoard
  'Draw the tic tac toe board: four straight lines
  TabletPen(_Black; 2)
  TabletLine(66, 3, 66, 196)
  TabletLine(133, 3, 133, 196)
  TabletLine(3, 66, 193, 66)
  TabletLine(3, 133, 196, 133)
END PROC

PROC DrawPiece(row, col, player)
  'Draw game piece: X for player 1, O for player 2
  LOCAL cx, cy
  cx = 33 + (col - 1)*66
  cy = 33 + (row - 1)*66
  IF player = 1 THEN
    TabletPen(_Blue; 5)
    TabletLine(cx-20, cy-20, cx+20, cy+20)
    TabletLine(cx-20, cy+20, cx+20, cy-20)
  ELSE
    TabletPen(_Red; 5)
    TabletEllipse(cx-20, cy-20, cx+20, cy+20)
  END IF
END PROC

PROC formproc(params)
  LOCAL row, col

  FormSelect(params[_FormNum])
  SELECT CASE params[_ItemNum]

  CASE 30  'Mouse click in the tablet
    'Compute the square clicked on
    col = Floor(TabletMouseX / 67) + 1
    row = Floor(TabletMouseY / 67) + 1

    'Is there already a piece there?

```

```

        IF Squares[row, col] THEN
            BEEP
            EXIT PROC
        END IF

        'Put a piece in the square
        Squares[row, col] = Player
        DrawPiece(row, col, Player)
        Player = 3 - Player

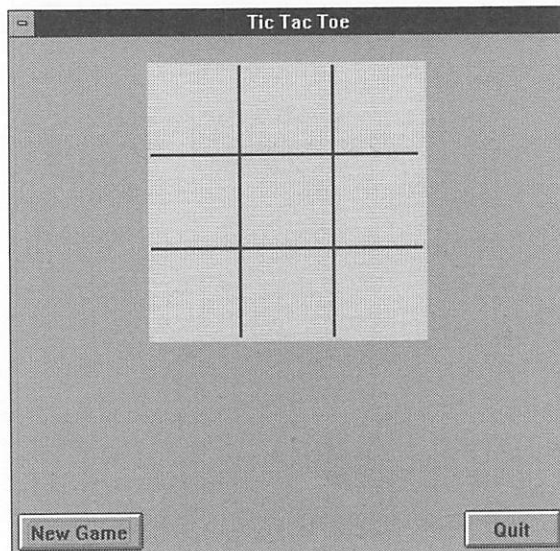
    CASE 10 'New Game
        TabletControl(_Clear)
        DrawBoard
        Squares[1:3, 1:3] = 0

    CASE 20 'Quit
        FormControl(_Close)
        EXIT PROGRAM
    END SELECT
END PROC

'Create the form with the tablet inside it
formTTT = FormQUnique
FormNew(formTTT; "Tic Tac Toe", _Title)
FormControl(_Size; _Center, _Center, 50 pct, 70 pct)
FormSetColor(_LightGray)
FormSetColor(_Yellow; _Field)
FormSetObject(10, _Button, "New Game", \
    _Left, _Bottom)
FormSetObject(20, _Button, "Quit", _Right, _Bottom)
FormSetObject(30, _Tablet, "", \
    _Center, 5 pct, 200 pxl, 200 pxl)
FormSetProc(formproc)

'Initialize the game
Player = 1
Squares[1:3, 1:3] = 0
DrawBoard
FormControl(_Show)

```

A Tablet in Action—Simple Draw!

The following program illustrates a number of features of the tablet in a ultra-simplified drawing program. It allows lines, rectangles, and ellipses to be drawn using any color. The rectangles and ellipses can be filled or empty.

The first click in the tablet defines the starting point of a line or the corner of an ellipse's bounding box or rectangle. The second click defines the second point of the line or the opposite corner of the bounding box or rectangle. The first point is shown with a single pixel that is removed when the second click has been made.

```
' Manual\PG_16\DRAW.RLZ

'Define constants for objects
CONST OBJ_LINE = 30
CONST OBJ_RECTANGLE = 40
CONST OBJ_ELLIPSE = 50
CONST OBJ_COLOR = 70
CONST OBJ_FILL = 80
CurrentObj = OBJ_LINE
FirstPt = 0
```

```
PROC formproc(params)
  FormSelect(params[_FormNum])
  SELECT CASE params[_ItemNum]
    CASE 10
      ProcessObject
    CASE OBJ_LINE, OBJ_RECTANGLE, OBJ_ELLIPSE
      CurrentObj = params[_ItemNum]
  END SELECT
END PROC

PROC ProcessObject
  IF Not(FirstPt) THEN
    'Save first point
    Pt1x = TabletMouseX
    Pt1y = TabletMouseY

    'Draw marker pixel
    ColorPt = TabletGetPixel(Pt1x, pt1y)
    TabletSetPixel(Pt1x, Pt1y, ColorPt+1)
    FirstPt = 1
  ELSE
    'Get Second point
    Pt2x = TabletMouseX
    Pt2y = TabletMouseY

    'Undraw marker pixel
    TabletSetPixel(Pt1x, Pt1y, ColorPt)
```

```

        'Set current colors
        TabletPen(FormQNum(OBJ_COLOR))
        IF (FormQNum(OBJ_FILL))
            TabletBrush(FormQNum(OBJ_COLOR))
        ELSE
            TabletBrush(_Empty)
        END IF

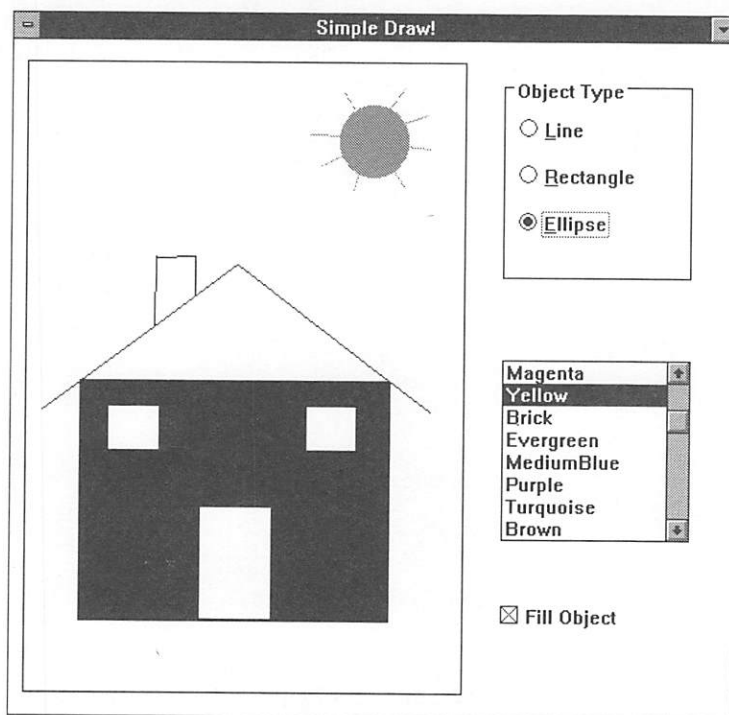
        'Draw object
        SELECT CASE CurrentObj
            CASE OBJ_LINE
                TabletLine(Pt1x, Pt1y, Pt2x, Pt2y)
            CASE OBJ_RECTANGLE
                TabletRectangle(Pt1x, Pt1y, Pt2x, Pt2y)
            CASE OBJ_ELLIPSE
                TabletEllipse(Pt1x, Pt1y, Pt2x, Pt2y)
        END SELECT
        FirstPt = 0
    END IF
END PROC

FormNew(;"Simple Draw!", _Close + _Title + \
_Minimize)
FormControl(_Size; 20 pct, 5 pct, 65 pct, 90 pct)
FormSetObject(10, _Tablet, "", 2 pct, 3 pct, \
61 pct, 94 pct)
FormSetObject(20, _GroupBox, "Object Type", \
68 pct, 5 pct, 26 pct, 30 pct)
FormSetObject(OBJ_LINE, _OptionButton, "&Line", \
70 pct, 10 pct; _Notify, 1)
FormSetObject(OBJ_RECTANGLE, _OptionButton, \
"&Rectangle", 70 pct, 17 pct; _Notify)
FormSetObject(OBJ_ELLIPSE, _OptionButton, \
"&Ellipse", 70 pct, 24 pct; _Notify)
FormSetObject(OBJ_COLOR, _ListBox, "", 68 pct, \
47 pct, 26 pct, 30 pct; ColorNames, \
_ListSelect, 1)
FormSetObject(OBJ_FILL, _CheckBox, "Fill Object", \
68 pct, 82 pct)
FormControl(_Show)
FormSetProc(formproc)

width = FormQObject(10)[_FQO_Width]
height = FormQObject(10)[_FQO_Height]
TabletRectangle(0, 0, width -1, height -1)

```

The following result can be created:



Chapter 17

The Animator

In This Chapter	17-1
How the Animator Works	17-2
Creating an Animation	17-3
Defining Cells	17-4
Defining Frames	17-6
Controlling the Animation	17-7
Starting and Stopping the Animation	17-7
Controlling the Animation Speed	17-8
Controlling the Animation Position	17-8
Setting Special Frames	17-8
Notification of Mouse Clicks	17-9
Using Pictures with Animation	17-10

Chapter 17

The Animator

In This Chapter

The Animator adds an exciting application tool to CA-Realizer. The Animator creates an animation sequence by displaying a series of bitmaps in succession. By controlling the order of the bitmaps, as well as the length of time they are displayed, you can create moving logos, fancy animated buttons, and even complete animated “movies.”

Like the graphics tablet, animations can only be created as an object within a form. Animations can be placed in a form with any of the standard CA-Realizer form objects, or with other programmable application tools. You can place multiple animation sequences in the same form.

Note: All the bitmaps and files for this chapter are located in the \REALIZER\MANUAL subdirectory.

The following table briefly describes the CA-Realizer commands and functions that allow you to use the Animation tool:

Command	Description
AnimateCells	Defines one or more bitmap cells for the currently selected animation window.
AnimateControl	Controls the current animation window.
AnimateFrame	Adds a frame to the end of the animation frame sequence for the current animation window.

Continued

Continued

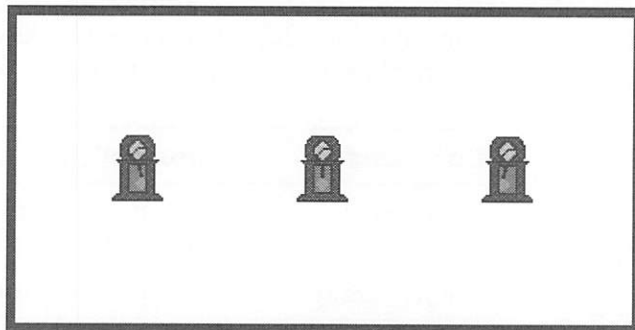
Command	Description
AnimateSelect	Selects the specified animation window as the current animation window. All subsequent animation commands will apply to this window.
AnimateSpecialFrame	Adds a special frame to the end of the animation frame sequence for the current animation window.

Many of these commands and functions are discussed in this chapter. For additional information, refer to the *CA-Realizer Language Reference Guide*.

How the Animator Works

True animation in film and video is created by drawing a set of pictures so that each one differs only slightly from the one before it. When these pictures, called *cells*, are shown in rapid succession, the picture appears to be moving.

Consider these three cells of a grandfather clock:



A complete forward movement consists of showing these three cells in order. For a full cycle, you can show the cells in the order: 1, 2, 3, 2. This ordering is called a sequence of *frames*.

A single frame consists of three parameters:

- The cell number to display
- The position at which to display it
- The length of time it should be displayed

If you use each cell only once in the animation sequence, the number of frames will be the same as the number of cells. However, if you use some of the cells more than once, as in the case of the clock, the number of frames will be greater than the number of cells.

Creating an Animation

To create an animation, all you need to do is specify the list of cells and the order in which you want to display them. The easiest way to create an animation is to have all the cells named in a sequence, such as CLOCK1.BMP, CLOCK2.BMP, and CLOCK3.BMP.

For example:

```
' Manual\PG_17\CLOCK1.RLZ

'Create a form with an animation window
FormNew( "Ticking", _Title + _Close)
FormControl(_SizeInside; _Center, _Center, \
    100 pxl, 100 pxl)
FormSetObject(10, _Animate, "", _Center, _Center, \
    100 pxl, 100 pxl)
FormControl(_Show)

'Define the three cells
AnimateCells("DEMOS\ANIMATE\BITMAPS\CLOCK#", 1, 3)

'Create a frame for each cell
AnimateFrame(1, 30, 30, 300)
AnimateFrame(2, 30, 30, 300)
AnimateFrame(3, 30, 30, 300)
AnimateFrame(2, 30, 30, 300)

'Start the animation
AnimateControl(_Start)
```

- AnimateCells** The program creates a form, resizes it, and places an animation window in the middle. Three cells are then defined. Since the file names of the bitmaps only differ by a number, we can replace the number with a pound sign (#) and tell **AnimateCells** what the first and last values for the number are.
- AnimateFrame** We need the cells to be played in the order 1, 2, 3, 2 to display a full motion. The **AnimateFrame** command is used once for each cell, and in each case the cell is drawn at location (30, 30) and held there for 300 milliseconds.
- AnimateControl(_Start)** The last step is to start the animation. This is accomplished with the **AnimateControl(_Start)** command. When the animation sequence is finished, it automatically restarts at the beginning.
- It is important to note that, although the clock is moving, no program is running. Therefore, you can create multiple instances of the moving clock at the same time. Try running this program a number of times without resetting the system. (Notice how they tend to synchronize with one another if you drag and hold one of the windows.)

Defining Cells

Each cell in the animation sequence is numbered. Use these numbers in the frames to indicate the cell to display. Define your cells with the **AnimateCells** command:

```
AnimateCells(Name [, CellNum])
```

Name is the name of the bitmap file for the cell and *CellNum* is the number of the cell you are defining. You do not have to define cells in order, and you can redefine a cell by using its number again with **AnimateCells**. The first format of the **AnimateCells** command creates a single cell.

You can also define an entire set of cells at once by specifying arrays for *Name* and *CellNum* parameters. If *CellNums* is not specified, the cells are numbered consecutively beginning at 1.

```
'Define cells 1 through 5
Names = {"CELL1.BMP", "CELL2.BMP", "CELL3.BMP", \
        "CELL4.BMP", "CELL5.BMP"}
AnimateCells(Names)
```

If the bitmap names differ only by a number, as they do in the above example, the second form of *AnimateCells* can be used. It also allows you to create multiple cells at one time.

```
AnimateCells(Format, Start, End [, FirstCellNum])
```

Format is a format string representing the names of the cells. It should include a single pound sign (#) in place of the digit or digits for the cell number. *Start* and *End* specify the range of values for the missing number in the format string:

```
AnimateCells("CELL#.BMP", 1, 5)
```

This command defines the same five cells as the previous example. By default, the cells are numbered according to the range. If the range for the format is 5 to 10, the cells defined will also be numbered 5 to 10. For the cell numbers to be different, specify the *FirstCellNum* parameter:

```
AnimateCells("CELL#.BMP", 1, 3, 17)
```

This would define three cells—CELL1.BMP as cell number 17, CELL2.BMP as 18, and CELL3.BMP as 19.

Defining Frames

AnimateFrame

An animation sequence is a series of *frames*. Each frame represents a cell shown at a specific position for a specific amount of time. You can create frames with the AnimateFrame command:

```
AnimateFrame(CellNum, XPos, YPos, Time)
```

This command adds a frame to the end of the animation frame sequence. The frame consists of cell number *CellNum* shown at position (*XPos*, *YPos*) for *Time* milliseconds.

The following program creates 60 frames that cause the clock to move around the perimeter of a circle while the pendulum swings.

```
' Manual\PG_17\CLOCK2.RLZ

FormNew( "Moving Clock", _Title + _Close)
FormControl(_SizeInside; _Center, _Center, \
    300 pxl, 300 pxl)
FormSetObject(10, _Animate, "", _Center, _Center, \
    300 pxl, 300 pxl)
FormControl(_Show)

AnimateCells("DEMOS\ANIMATE\BITMAPS\CLOCK#", 1, 3)
Cells[0:3] = {1,2,3,2}
n = 0
FOR t = 1 to 360 step 6
    Theta = t * _Pi / 180
    AnimateFrame(Cells[n Mod 4], 120 + 100 * \
        Cos(Theta), 120 + 100 * Sin(Theta), 100)
    n = n + 1
NEXT t
AnimateControl(_Start)
```

Controlling the Animation

The Animator uses a more restricted version of the *TOOLControl* command to control the animation window:

```
AnimateControl(Action [, Mod1 [, Mod2]])
```

The following table lists the valid *Action* constants:

Constant	Description
<code>_Start</code>	Starts the animation sequence.
<code>_Stop</code>	Stops the animation sequence.
<code>_Reset</code>	Starts the animation sequence at the beginning.
<code>_Clear</code>	Clears part or all of the data in the animation window. If <i>Mod1</i> is <code>_Frame</code> , CA-Realizer only clears the frame sequence. If <i>Mod1</i> is <code>_All</code> or unspecified, CA-Realizer clears both the frame sequence and the cell definitions.
<code>_SetSpeed</code>	Sets the speed factor to <i>Mod1</i> .
<code>_SetOffset</code>	Sets the position offset to (<i>Mod1</i> , <i>Mod2</i>).

Starting and Stopping the Animation

After the animation sequence of cells and frames has been defined, it can be set into motion with the `AnimateControl(_Start)` command; it can then be stopped with `AnimateControl(_Stop)`.

If the sequence is started again after it has been stopped, it will continue with the frame at which it left off. To restart the sequence from the beginning, use `AnimateControl(_Restart)`.

Controlling the Animation Speed

The length of time each frame is shown depends on the *Time* value specified when the frame was defined and on the animation *speed factor*, defined as follows:

```
AnimateControl(_SetSpeed; SpeedFactor)
```

When the animation is running, the two values are multiplied together to determine how long the frame should be displayed. The speed factor is initially 1, so the durations are exactly those specified in the frame definitions. The speed factor (*SpeedFactor*) can be set to any positive integer value with the `AnimateControl(_SetSpeed)` command.

Controlling the Animation Position

The position at which the cell is displayed within the animation window is defined for each frame with `AnimateControl`:

```
AnimateControl(_SetOffset; X, Y)
```

These values are actually added to an offset position that is initially (0, 0)—the offset position can be changed to (X, Y). The entire animation window can also be moved by using the `FormModifyObject` command and specifying new values for *Left* and *Top*.

Setting Special Frames

Three special frames can be inserted into the animation sequence using the `AnimateSpecialFrame` command:

```
AnimateSpecialFrame(_Pause; Delay)
AnimateSpecialFrame(_Stop)
AnimateSpecialFrame(_Notify)
```

`_Pause` stops the animation for *Delay* milliseconds without anything being displayed, `_Stop` forces the animation loop to halt, and `_Notify` invokes the form procedure for the form containing the animation window with a `_Click` message.

Notification of Mouse Clicks

An animation window notifies its form with a `_Click` message when either the user clicks the mouse button while pointing to the window or when a notify frame is reached in the sequence. Notification causes a `FormWait` command to return or the form procedure to be invoked.

Two types of notification are possible using the `FormQNum` function inside the form procedure. The value will either be `_Click` if the user clicked inside the animation window, or `_Notify` if a notify frame was reached.

For mouse clicks, the position of the click can be found in the `params[_XPos]` and `params[_YPos]` values of the parameter array. Refer to “Overview of the CA-Realizer Programmable Application Tools” in this guide for more information.

The following form procedure allows the animation to be toggled on and off when the user clicks in the window:

```
MovingFlag = 1

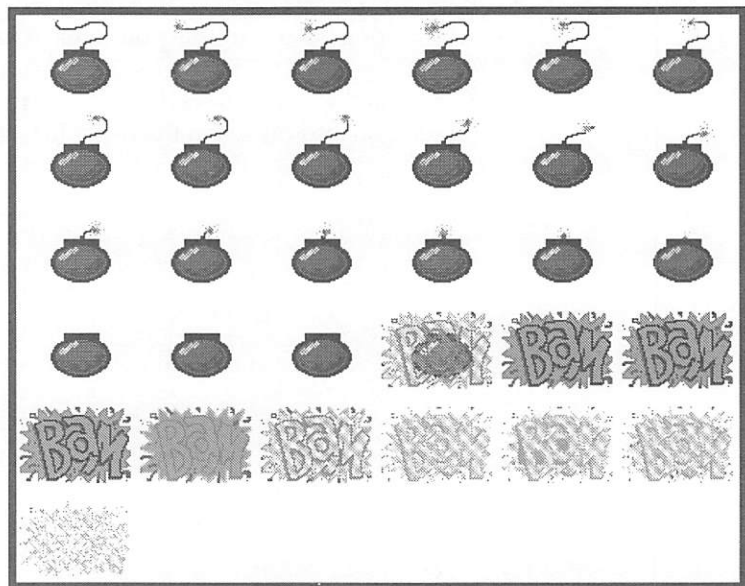
PROC formproc(params)
  FormSelect(params[_FormNum])
  SELECT CASE params[_ItemNum]
    CASE 20  'Animation Window
      AnimateSelect(20)
      IF MovingFlag THEN
        AnimateControl(_Stop)
      ELSE
        AnimateControl(_Start)
      END IF
      MovingFlag = Not(MovingFlag)
    END SELECT
  END PROC
```

Using Pictures with Animation

Lengthy animation sequences require a large number of bitmaps for the cells. If each of the cells is stored as a separate file, they will take up a larger amount of disk space when they are distributed and installed. Extracting cells from a large bitmap using picture resources can reduce these problems dramatically.

The idea is to combine some or all of the bitmaps into a single bitmap as a row or grid of cells. A picture is then created from this bitmap, and the cell pictures are created as regular portions of the first picture.

Consider the following example—it contains 31 frames of an exploding bomb:



This form was created with the following program:

```
' Manual\PG_17\BOMBPIC1.RLZ
'Determine the size of each bitmap
FormNew
FormSetObject(999, _Bitmap,
"DEMOS\ANIMATE\BITMAPS\BOMB1.BMP", 0, 0)
width = FormQObject(999)[_FQO_Width]
height = FormQObject(999)[_FQO_Height]
FormModifyObject(999, _Close)

'Size the form for a 6x6 grid of bitmaps
FormControl(_SizeInside; _Center, _Center, \
width * 6 pxl, height * 6 pxl)
FormControl(_Show)

FOR n = 1 to 31
  row = (n - 1) \ 6
  col = (n - 1) MOD 6
  x = col * width
  y = row * height
  FormSetObject(n, _Bitmap, \
    Sprint("DEMOS\ANIMATE\BITMAPS_\
    BOMB(0)", n), x, y)
NEXT n
```

First, the program creates a form, loads a bitmap, and determines its size. The inside of the form is resized to fit a 6 x 6 grid of the 31 pictures, and then the FOR loop cycles through the bitmaps, determines where each belongs in the matrix, and displays them.

Once created, this form can be copied to the Clipboard with the EDIT COPY WINDOW menu command. It can then be pasted into a drawing program and saved as a single bitmap image.

The following program uses the combined bitmap in an animation window, extracting each cell using similar logic:

```
' Manual\PG_17\BOMBPIC2.RLZ

'Create a picture for the combined bitmap
picBomb = PictureNew(; "BOMBS.BMP")

'Create a form
FormNew(; "Bomb!", _Title + _Close)

'Compute the size of each cell, resize the form
' and create the animation window
FormSetObject(999, _Bitmap, picBomb, 0, 0)
width = FormQObject(999)[_FQO_Width] / 6
height = FormQObject(999)[_FQO_Height] / 6
FormModifyObject(999, _Close)
```

```
FormControl(_SizeInside; _Center, _Center, \\  
    width px1, height px1)  
FormSetObject(20, _Animate, "", 0 pct, 0 pct, \\  
    100 pct, 100 pct)  
FormControl(_Show)  
  
'Create a picture and an animation cell for each  
FOR n = 1 to 31  
    row = (n - 1) \ 6  
    col = (n - 1) MOD 6  
    x = col * width  
    y = row * height  
    PictureNew(100+n; picBomb, _Bitmap, x, y, \\  
        width, height)  
    AnimateCells(100+n, n)  
NEXT n  
  
'Create the frames  
AnimateFrame(1, 0, 0, 250)  
FOR n = 2 to 31  
    AnimateFrame(n, 0, 0, 50)  
NEXT n  
AnimateSpecialFrame(_Pause; 1000)  
  
'Start the animation  
AnimateControl(_Start)
```

In this case, the combined bitmap is loaded as a picture and its size is computed. Since the cells are stored as a 6 x 6 grid, the width and height of each cell is 1/6 of the width and height of the larger bitmap.

The inside of the form is resized to the size of a cell, and an animation window is placed in the form. After extracting each cell using a method similar to the one used for positioning them, the program then creates the cell as a picture and an animation cell.

This method has a number of advantages. First, since only one bitmap has to be loaded from disk, the total time for creating the animation sequence is reduced. Additionally, although the single bitmap uses the same amount of memory after it is loaded, it consumes fewer system resources, is easier to distribute, and takes up less room on a hard drive or a floppy disk.

Chapter 18

Custom Menus and Accelerator Keys

In This Chapter	18-1
The Anatomy of a Menu	18-1
Creating a Menu	18-3
Generating an ID	18-3
Specifying an ID	18-4
Creating the Menu	18-4
Specifying a Keyboard Accelerator	18-4
Selecting a Menu	18-5
Controlling a Menu	18-6
Displaying a Menu	18-7
Specifying a Location	18-7
Hiding CA-Realizer's Menus	18-8
Controlling Menu Redraw	18-9
Repositioning Menus	18-9
Closing a Menu	18-9
Adding Items to a Menu	18-10
Custom Menu Commands	18-10
Separators	18-11
Modifying Menu Items	18-12
Setting a Menu Procedure	18-14
Attaching a Menu Procedure	18-14
A Sample Menu Procedure	18-15

Using CA-Realizer Menu Commands in Your Menus	18-17
Adding Submenus to a Menu	18-18
Creating a Submenu	18-19
Adding Items to a Submenu	18-20
Using Accelerators	18-21

Chapter 18

Custom Menus and Accelerator Keys

In This Chapter

In this chapter, you will learn how to create a set of custom menus for your CA-Realizer applications. You will also learn how to link menus and accelerators to procedures in a CA-Realizer program.

The Anatomy of a Menu

Menu

A *menu* is a user interface element that presents the user with a list of command choices. Menus appear at the top of the application window in the *menu bar*.

Every menu has a *title* (or *name*) that appears in the menu bar. For example, the CA-Realizer menu bar includes a FILE menu and an EDIT menu, as well as several other menus.

Menu Items

Additionally, each menu contains one or more *items*. (A menu item can be a command, a separator, or a submenu, all of which are described in this chapter.) For example, the CA-Realizer FILE menu contains several commands and separators.

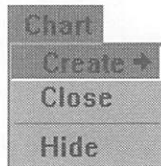
When a menu is selected by the user (either with the mouse or the keyboard), it displays a drop-down menu that lists all of the items defined to the menu. You can then select a menu command from this list (using either the mouse or the keyboard).

Here is a sample menu—its title or name is **STYLE**, and its drop-down menu contains four menu commands and one separator:

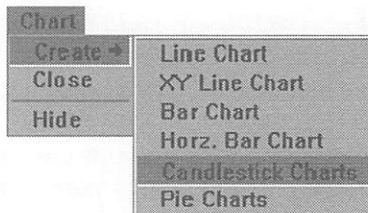


As pointed out above, menu commands can have different *states*—normal, checked, and grayed. They are described later in this chapter. Accelerator keys and separators are also discussed in greater detail later in the chapter.

Note that a submenu, like the one below, always appears with an arrow at the right side of its name to indicate that, when selected, it displays another menu. For example, the **CHART** menu below contains a submenu named **CREATE**:



When the user selects **CREATE**, it displays the following:



Note: Submenus can be nested up to eight levels deep.

Creating a Menu

CA-Realizer has several built-in menus. When creating a CA-Realizer application, you can temporarily remove the CA-Realizer menus from the menu bar, or you can mix them with your own menus and menu items.

Use the `MenuNew` command to create a menu. When creating a menu, you can either have CA-Realizer generate a unique menu number (ID) for you, or you can specify a particular menu ID.

You then use this menu ID with any of the commands that accept a menu as a parameter (for example, `MenuSelect` and `MenuControl`).

Generating an ID

MenuQUnique

You can use the `MenuQUnique` function before issuing the `MenuNew` command to generate a unique menu ID. In this example, `MenuQUnique` generates a new ID and assigns it to the *MenuID* variable. It is then used in the next `MenuNew` command:

```
MenuID = MenuQUnique  
MenuNew(MenuID; "Style")
```

MenuNew

You can also instruct the `MenuNew` command to generate a unique ID. Omitting the optional *MenuNum* parameter or by setting it to 0, will generate a unique menu ID automatically. (See [Creating the Menu](#) below for details on the syntax of `MenuNew`.)

Specifying an ID

To explicitly set a particular menu ID, specify the desired number as the optional *MenuNum* parameter in the *MenuNew* command. (See *Creating the Menu* below for details on the syntax of *MenuNew*.)

Menu IDs can be any number between 1 and 32767. If you specify an ID for a new menu that is already used by an existing menu, the old menu is destroyed and the specified ID is set for the new menu. For example, creating a Menu 10 destroys any previous Menu 10.

Creating the Menu

The general syntax of the *MenuNew* command is:

```
[MenuID = ] MenuNew(MenuNum [, ParentNum]; Title)
```

MenuID is the variable to which the new menu's ID is optionally assigned. *MenuNum* is the ID to be set for the new menu. (If *MenuNum* is 0 or omitted, a unique menu number is generated automatically and returned as *MenuID*.)

ParentNum is used when creating a submenu—please refer to *Adding Submenus to a Menu* in this chapter for details.

Title is the name of the menu that will be displayed in the application's menu bar.

Specifying a Keyboard Accelerator

When setting the *Title* of a new menu, you can place an ampersand before a letter in the specified menu name to create an *accelerator* for that menu.

An accelerator is a standard GUI convention used to select a menu from the keyboard. It is simply an underlined letter in a menu name. For example, the CA-Realizer FILE menu uses "F" as its accelerator.

After setting an accelerator for a menu command, you can press ALT+underlined letter to select the menu. (Otherwise, the user must click the menu name with the mouse.)

As stated above, to set a particular letter to be a menu's accelerator, place an ampersand before the desired letter when specifying the name in the MenuNew command. For example:

```
MenuID = MenuQUnique  
MenuNew(MenuID; "&Style")
```

Now the letter "S" will be underlined when it is displayed in the menu bar. Additionally, pressing ALT+S would select this new menu.

Note: If an ampersand is needed in the menu title, you can use &&.

Selecting a Menu

By default, when you create a menu with MenuNew, it becomes the current menu (this means that **all** subsequent menu commands apply to it). To select a different current menu, use MenuSelect:

```
MenuSelect(MenuID)
```

MenuID is the ID of the menu that should now be the current menu. You can also specify one of the following constants to select one of CA-Realizer's menus: `_FileMenu`, `_EditMenu`, `_RunMenu`, `_WindowMenu`, and `_HelpMenu`.

For example:

```
MenuSelect(10)  
MenuSelect(_FileMenu)
```

Controlling a Menu

Once you create a menu, you control its appearance on the menu bar with the `MenuControl` command:

```
MenuControl(Action [; Pos])
```

The valid values for *Action* are listed below:

Action	Description
<code>_Close</code>	Closes and destroys the menu, removing it from the menu bar or a submenu's parent menu.
<code>_Hide</code>	Hides the menu and removes it from the menu bar. You cannot use this command on submenus.
<code>_SetRedraw</code>	Controls whether changes to the menu are displayed immediately. If <i>Pos</i> is 1, changes are displayed immediately. If <i>Pos</i> is 0, changes are not displayed until the redraw is set back to 1.
<code>_Show</code>	Displays the menu in the menu bar. By default, CA-Realizer adds the menu to the end of the menu bar. If you specify <i>Pos</i> , CA-Realizer places the new menu after the menu with a menu number of <i>Pos</i> . If <i>Pos</i> is 0, CA-Realizer places the new menu at the beginning of the menu bar. You cannot use this command on submenus.

Pos controls the location of the menu on the menu bar. It can be either a valid menu number or one of the constant values representing the built-in menus.

Note: Any menu you create remains in the CA-Realizer menu bar until one of the following occurs:

- The menu is explicitly hidden.
- The menu is closed with the `MenuControl` command.
- The menu is reset using `CTRL+ALT+F2` or the `RESET _Menu` command.

Displaying a Menu

When a menu is created, it is invisible. The `MenuControl` command can be used to make it visible. (Like most tool-related commands, `MenuControl` acts on the current menu.)

For example, the following statement

```
MenuControl(_Show)
```

displays the current menu in the menu bar.

Specifying a Location

By default, a new menu is added to the end (right) of the menu bar, after all the standard CA-Realizer menus.

To display a menu at a particular location, specify a modifier in the `MenuControl(_Show)` command that instructs CA-Realizer where to display the menu. The following is the general syntax:

```
MenuControl(_Show; Pos)
```

where *Pos* is either: the ID of an existing menu after which the new menu should be displayed, a value of 0 (to place the new menu at the beginning (far left) of the menu bar), or one of the following constants, representing the CA-Realizer menu after which the new menu should be displayed: `_FileMenu`, `_EditMenu`, `_RunMenu`, `_WindowMenu`, or `_HelpMenu`.

For example:

```
'Add a menu to the end of the menu bar
MenuNew(10; "&MyMenu")
MenuControl(_Show)
```

```
'Add a menu after the CA-Realizer Edit menu
MenuControl(_Show; _EditMenu)
```

Hiding CA-Realizer's Menus

The standard CA-Realizer menus will remain in your application's menu bar unless explicitly hidden. You can use the `MenuControl` command to hide them, which allows you to create your own customized menu bar or to replace the standard CA-Realizer menus with your own.

You can hide the current menu on the menu bar with:

```
MenuControl(_Hide)
```

When you hide a menu, it disappears from the menu bar—the other menus shift to the left to fill the gap.

You can hide all of the CA-Realizer menus *simultaneously*, by using the following statements:

```
FOR menu = -5 to -1
  MenuSelect(menu)
  MenuControl(_Hide)
NEXT menu
```

Note that hiding a menu does not destroy its contents. A hidden menu can still be referenced by its menu ID, and can be redisplayed at any time using the `MenuControl(_Show)` command.

You can restore the built-in menus of the CA-Realizer menu bar and remove all custom menus by resetting the environment with `CTRL+ALT+F2`.

Note: While you can hide CA-Realizer's standard menus, you cannot close them.

Controlling Menu Redraw

If a number of menu items are being changed at the same time, it can be useful and faster to prevent the changes from being shown until they are all complete. In such a case, execute the following statement

```
MenuControl(_SetRedraw; 0)
```

to prevent changes from being shown immediately. You can use `MenuControl(_SetRedraw; 1)` to restore normal updating.

Repositioning Menus

To reposition a menu after you place it in the menu bar, you must first hide it, and then redisplay it. For example, assuming you have two menus defined as `BOARD` and `RUN`, the following commands move the `BOARD` menu after the `RUN` menu:

```
MenuSelect(BoardMenu)
MenuControl(_Hide)
MenuControl(_Show; _RunMenu)
```

Closing a Menu

To destroy a menu and its contents, use the `MenuControl(_Close)` command.

When you close a menu, CA-Realizer removes it from the menu bar. You can then reuse its menu number for other menus, without conflict.

Note: You can never close CA-Realizer's system menus. The only way to temporarily remove built-in menus from the menu bar is to hide them.

Adding Items to a Menu

Once you have created a menu, you can add items—such as menu commands, separators, and/or submenus—to it. Note that you can create your own custom menu commands or you can add some of CA-Realizer's menu commands to your own custom menus.

This section describes how to create and add your own menu commands and separators to a menu. Incorporating CA-Realizer menu commands and submenus is described later in this chapter in the Using CA-Realizer Menu Commands in Your Menus and Adding Submenus to a Menu sections, respectively.

Both custom menu commands and separators are created using the `MenuSetCmd` command. The general form of `MenuSetCmd` is as follows:

```
MenuSetCmd(ItemID [, Text] [, Key] [, Style])
```

Note that when `MenuSetCmd` is executed, it adds the specified command to the current menu. Additionally, menu items appear on the menu in the order in which you add them.

Custom Menu Commands

A *menu command* is an item that when chosen from its parent menu by the user, performs an action, like displaying a dialog box or toggling a condition on or off.

To add a custom menu command to a menu, specify *ItemID* in `MenuSetCmd` as a number between 1 and 32,767. This number must be unique only to the menu.

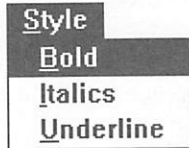
Additionally, you specify *Text* to indicate the menu command's title or name (this is the text that is displayed on the menu). Like menus, you can use an ampersand (&) to set an accelerator key for the menu item (the character following the & is underlined).

For example:

```
MenuID = MenuQUnique
MenuNew(MenuID; "&Style")

MenuSetCmd(1, "&Bold")
MenuSetCmd(2, "&Italics")
MenuSetCmd(3, "&Underline")

MenuControl(_Show)
```



In `MenuSetCmd`, *Key* specifies a single key accelerator to use for the menu item. If *Key* is given for a menu item without a *Text* parameter, the accelerator is active, but the command does not appear in the menu. (For more information about single key accelerators, and for a complete list of valid *Key* parameters, see Using Accelerators later in this chapter.)

Style controls the appearance of a menu command. For example, you may want to create a menu command that is *checked* to indicate that a certain condition is active. Menu command styles are described in greater detail in the Modifying Menu Items section later in this chapter.

Note: By default, menu commands are created as unchecked and ungrayed.

Separators

A *separator* is a thin line that can be placed on a menu to group similar commands together. For example, if a menu contained two distinct sets of commands, a separator could be added to visually distinguish these two groups of commands.

To add a separator to a menu, simply specify the *ItemID* parameter as `_Separator`. For example:

```
MenuSetCmd(_Separator)
```

Modifying Menu Items

Once you have added a command to a menu, you can change it at any time by using `MenuSetCmd` with that command's ID number. For example, you can alter a command's title by specifying a new value for *Text*. Since the menu command specified by the given ID already exists, CA-Realizer preserves all the other attributes specified for it.

You can also change the appearance of menu commands by *checking/unchecking* or *graying/enabling* them.

Checked Commands While many menu commands are used to display a dialog box or perform an action, some are used to indicate a *toggle setting*. A toggle setting is used to indicate an ON/OFF condition.

The state of the toggle setting command is usually indicated to the user with a check mark, which does or does not appear depending on whether the toggle setting is on or off. A check mark is displayed on a menu to the left of a specified menu command.

Check marks are appropriate for indicating a switch among several possible states. For example, a menu for selecting a font style can contain commands for italicizing, bolding, and underlining text. By placing a check mark next to one of these commands, you can indicate the currently active font style.

Try running the following program to view how checked commands look:

```
SizeMenu = MenuQUnique
MenuNew(SizeMenu; "&Size")

MenuSetCmd(120, "Small"; _Check)
MenuSetCmd(130, "Medium")
MenuSetCmd(140, "Large")

MenuControl(_Show)
```



Note: To restore a checked menu command to its normal state, execute `MenuSetCmd` again, using `_UnCheck` as a modifier.

Tip: Commands can also be toggled by changing their text with the `MenuSetCommand`. For example, the `GO` command in the `CA-Realizer RUN` menu changes to `STOP` when a program is running.

Grayed Commands

Grayed menu commands represent choices that are currently not available, and remain unavailable until enabled by the program. They let the user know that the choices exist, but that you cannot choose them at the moment. (For example, the `SAVE` command in the `CA-Realizer FILE` menu is grayed when the active window is not a program window or a log.)

Graying an item is useful for disqualifying specific program options when certain conditions exist.

To gray two commands, try the following:

```
SizeMenu = MenuQUnique
MenuNew(SizeMenu; "&Size")

MenuSetCmd(120, "Small")
MenuSetCmd(130, "Medium"; _Gray)
MenuSetCmd(140, "Large"; _Gray)

MenuControl(_Show)
```



Note: To restore a grayed menu command to its normal state, execute `MenuSetCmd` again, using `_Normal` as the style modifier.

Removing Items

You can also remove existing menu items at any time with the `_Remove` modifier of `MenuSetCmd`.

For example, to remove a menu number 80 from the current menu:

```
MenuSetCmd(80; _Remove)
```

Note: The remaining commands shift upwards to fill the space of the deleted menu command.

Setting a Menu Procedure

When the user selects an item from a menu, the procedure associated with that menu is invoked.

Tip: See the “Overview of the CA-Realizer Programmable Application Tools” chapter for information about tool procedures.

Attaching a Menu Procedure

Use `MenuSetProc` to associate a procedure or program module with the current menu:

```
MenuSetProc [(ProcName | ModuleName)]
```

ProcName should be the name of a defined procedure that takes one parameter (the menu procedure parameter array).

Note: `MenuSetProc` without any parameters disassociates the current menu procedure from the menu.

The menu procedure parameter array passed to the menu procedure contains information about the menu item that was selected. Specifically, *params[_MenuNum]* is the ID of the menu from which an item was selected and *params[_ItemNum]* is the ID of the item that was selected from that menu.

The general structure of a menu procedure is as follows:

```
PROC MyMenuProc(params)
  MenuSelect(params[_MenuNum])
  SELECT CASE params[_ItemNum]
  CASE 10
    'Action for menu item 1
  CASE 20
    'Action for menu item 2
  'Move cases
  END SELECT
END PROC
```

You can use the same menu procedure for more than one menu. If all the items in all the menus have different item numbers, the program just shown will work fine. If more than one menu has an item with the same item number, you will need an extra SELECT statement to first choose the menu and then to choose the item within it.

A Sample Menu Procedure

The program below creates and processes a new menu called SAMPLE. There are four items in the menu:

- When NORMAL is selected, it displays a message.
- When CHECKABLE is selected, a check mark is added or removed from the item.
- When the START command is selected, it changes to STOP and grays the GRAYED WHEN START item.
- If STOP is then selected, it changes back to START and the last item is ungrayed.

```
' Manual\PG_18\MENUPROC.RLZ

PROC MyMenuProc(params)
  MenuSelect(params[_MenuNum])
  SELECT CASE params[_ItemNum]
    CASE 1    'Normal
      INPUT "Aren't menus fun!";
    CASE 2    'Checkable
      IF Checked THEN
        MenuSetCmd(2; _Uncheck)
        Checked = 0
      ELSE
        MenuSetCmd(2; _Check)
        Checked = 1
      END IF
    CASE 3    'Start/Stop
      IF Started THEN
        MenuSetCmd(3, "&Start")
        MenuSetCmd(4; _Normal)
        Started = 0
      ELSE
        MenuSetCmd(3, "&Stop")
        MenuSetCmd(4; _Gray)
        Started = 1
      END IF
    CASE 4    'Grayed when Start
      INPUT "This grays when Start is selected";
  END SELECT
END PROC

SampleMenu = MenuUnique
MenuNew(SampleMenu; "&Sample")
MenuSetCmd(1, "&Normal")
MenuSetCmd(2, "&Checkable")
MenuSetCmd(_Separator)
MenuSetCmd(3, "&Start")
MenuSetCmd(4, "&Grayed when Start")
MenuSetProc(MyMenuProc)

Checked = 0
Started = 0
MenuControl(_Show)
```

Using CA-Realizer Menu Commands in Your Menus

To create a custom menu command with the same functionality as a standard CA-Realizer menu command, specify the *ItemID* parameter in `MenuSetCmd` as one of the predefined CA-Realizer menu constants.

For example, you can specify `_RLZM_OPEN` to execute the CA-Realizer FILE OPEN menu command or `_RLZM_PASTE` to execute the CA-Realizer EDIT PASTE menu command.

The following program, for example, uses CA-Realizer's standard COPY, CUT, and PASTE menu commands (from the EDIT menu) for use with the same-named commands on a custom menu titled "CUT & PASTE":

```
CutMenu = MenuQUnique
MenuNew(CutMenu; "Cut && &Paste")

MenuSetCmd(_RLZM_CUT, "C&ut")
MenuSetCmd(_RLZM_COPY, "&Copy")
MenuSetCmd(_RLZM_PASTE, "&Paste")

MenuControl(_Show)
```

The available constants are the same as those that are supplied for use with the `MenuDoCmd` command. Specifying such a constant simulates an action from one of the CA-Realizer menus. It provides actions not normally available through other commands.

Note: For a complete list and description of all available constants, see `MenuDoCmd` in the *CA-Realizer Language Reference Guide*.

The ability to incorporate CA-Realizer's system menu items into your custom menus is a very powerful feature. However, note the following:

- When the user selects the menu command, a user-defined menu procedure is not called, but rather the normal action for the CA-Realizer menu item is executed.
- The menu command also automatically grays in the same way that the CA-Realizer menu command does (that is, a CUT command that you create using `_RLZM_CUT` is grayed when there is no selected text). To avoid confusion, the original CA-Realizer menu containing this command should be hidden.

Adding Submenus to a Menu

Submenus have a title and appear on a menu like normal menu commands, except that each is displayed with an arrow to the right of its title to indicate that it is a submenu. When selected by the user in an application, a submenu displays another menu, providing a means to nest your application's menus.

Note: Submenus can be nested up to eight levels deep.

When you create a new submenu, it becomes the current menu, and subsequent `MenuSetCmd` calls add items to the new submenu. (Additional items or submenus can be added to the parent menu by first reselecting it with the `MenuSelect` command.)

Creating a Submenu

Submenus are created in almost the same way as top-level menus. The only difference is that an additional parameter, *ParentNum*, must be passed to `MenuNew` to indicate the number of the menu in which the submenu will be added.

To add a submenu to a menu, use the `MenuNew` command as follows:

```
[MenuID = ]MenuNew(MenuNum [, ParentNum]; Title)
```

Similar to when creating a menu, *MenuID* is the variable to which the new submenu's ID is optionally assigned and *MenuNum* is the ID to be set for the new submenu. (If *MenuNum* is 0 or omitted, a unique menu number is generated automatically and returned to *MenuID*.)

ParentNum is the ID of an existing top-level menu or submenu to which the submenu will be added. (This menu will act as the parent of the new submenu.) The value specified for *MenuNum* must be greater than *ParentNum*.

Additionally, *Title* is the name of the submenu that will be displayed as the parent menu's item.

For example:

```
'Create a new menu called Style
MenuID = MenuQUnique
MenuNew(MenuID; "&Style")

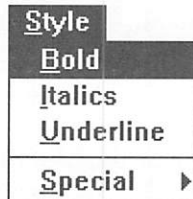
'Add menu commands
MenuSetCmd(1, "&Bold")
MenuSetCmd(2, "&Italics")
MenuSetCmd(3, "&Underline")

'Add a separator
MenuSetCmd(_Separator)
```

```
'Add a submenu
SubMenuID = MenuQUnique
MenuNew(SubMenuID, MenuID; "&Special")

'Select the top-level menu to display it
MenuSelect(MenuID)

MenuControl(_Show)
```



Adding Items to a Submenu

Once you have created a submenu, you can add other menu items or submenus to it. You can add menu commands, separators and other submenus (up to eight levels) to a submenu. (Be sure the desired submenu is selected before you start to execute new MenuSetCmd statements.)

Let us add submenu items to the SPECIAL submenu created in the previous section:

```
' Manual\PG_18\SUBMENUS.RLZ

'Create a new menu called Style
MenuID = MenuQUnique
MenuNew(MenuID; "&Style")

'Add menu commands
MenuSetCmd(1, "&Bold")
MenuSetCmd(2, "&Italics")
MenuSetCmd(3, "&Underline")

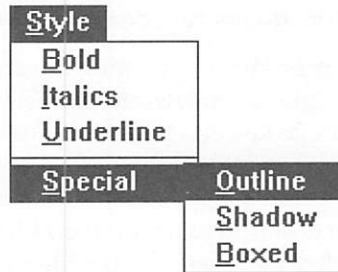
'Add a separator
MenuSetCmd(_Separator)

'Add a submenu
SubMenuID = MenuQUnique
MenuNew(SubMenuID, MenuID; "&Special")
```



```
'Add items to the submenu while it is still selected
MenuSetCmd(11, "&Outline")
MenuSetCmd(12, "&Shadow")
MenuSetCmd(13, "&Boxed")
```

```
'Select the top-level menu to display it
MenuSelect(MenuID)
MenuControl(_Show)
```



Using Accelerators

A menu command that appears on a menu can be selected by the user with the mouse (by clicking the menu and then the desired command) or with the keyboard.

If selecting a menu command with the keyboard, the user can use the standard GUI convention of ALT+underlined letter. For example, underlining the “F” in FILE and the “O” in OPEN allows the user to press ALT+F+O to choose the OPEN menu command from the FILE menu in CA-Realizer. (This was also described in the Custom Menu Commands section in this chapter.)

Note: If you use a standard CA-Realizer menu item in your custom menu, you cannot redefine its accelerator key.

You can also allow the user to invoke more frequently-used menu commands with a single keystroke by associating a menu command with a *single key* accelerator. For example, in CA-Realizer, in addition to being able to press ALT+F+O to choose the OPEN menu command from the FILE menu, you can use its single key accelerator, which is CTRL+F12.

Single key accelerators differ from ALT key accelerators in two ways:

- Single key accelerators are only activated when the user presses a designated accelerator key while there are no active menus. You can activate a menu by clicking on it or using an ALT accelerator. If a menu is active, single key accelerators do not function until you deactivate the menu.
- A menu procedure containing the appropriate action code for the single key accelerators must be attached to the menu for the single key accelerators to function.

To specify a single key accelerator for a menu command, you must include descriptive text in the *Title* parameter, as well as the optional *Key* parameter, in the `MenuSetCmd` command used to create the menu item.

For example, the following displays the text “F2” and “F3” to the right of the two menu commands defined to the `OPTIONS` menu and sets them as the single key accelerators for those commands:

```
MenuNew(0; "&Options")
MenuSetCmd(120, "&Statistics F2", _Virtual + \\  
_VK_F2)
MenuSetCmd(130, "Preferences F3", _Virtual + \\  
_VK_F3)
MenuControl(_Show)
```



These statements accomplish two things—they indicate the available single key accelerators to the user by displaying them on the menu and also define them to the system.

As displayed above, only a single space separates an accelerator's text from its menu command's name when separated by a single space in the program.

You can also right-justify accelerators on a menu by placing a tab between the menu command's name and its accelerator in the *Title* parameter text in the program.

For example:

```
MenuNew(0; "&Options")
MenuSetCmd(120, "&Statistics F2", _Virtual + \\  
_VK_F2)
MenuSetCmd(130, "Preferences F3", _Virtual + \\  
_VK_F3)
MenuControl(_Show)
```

<u>O</u> ptions	
<u>S</u> tatistics	F2
P <u>r</u> eferences	F3

Note: All available Key constants are provided in the *CA-Realizer Language Reference Guide*.)

Important! Once you add a single key accelerator, the key no longer performs its normal function. For example, if you use the `_VK_Delete` accelerator, the `DELETE` key will not operate in logs and edit controls.

Note: If an accelerator key is given for a command that has no text, the accelerator will be active but the command will not appear in the menu.

Chapter 19

The Scheduler

In This Chapter	19-1
The Anatomy of the Scheduler	19-2
Invoking the Scheduler	19-3
How Scheduled Commands are Run	19-4
Scheduling an Event	19-4
Repeating an Event	19-7
Starting and Stopping the Scheduler	19-8
Removing an Event from the Scheduler	19-9
Querying the Scheduler and Its Contents	19-10

Chapter 19

The Scheduler

In This Chapter

With the CA-Realizer Scheduler, you can run programs and commands automatically at specified times and intervals.

CA-Realizer's Scheduler allows you to store and execute events. These events consist of a command statement or a procedure name, and a date-time value. When the date and time specified for an event is reached, CA-Realizer executes the corresponding command. You can either schedule an event to occur only once, or it can repeat at regular intervals.

You can place any single line program statement in the Scheduler. For instance, you can use an assignment operation statement, a PRINT command, or a procedure call. You can use an event to retrieve live stock data from a wire service, to read data from an appointment book file, or to back up a data file.

The following table briefly describes the CA-Realizer commands and functions that allow you to create and control the Scheduler:

Command	Description
SchedControl	Controls the appearance and operation of the Scheduler.
SchedNew	Determines the appearance of the Scheduler window.
SchedQ	Returns information about the Scheduler window.

Continued

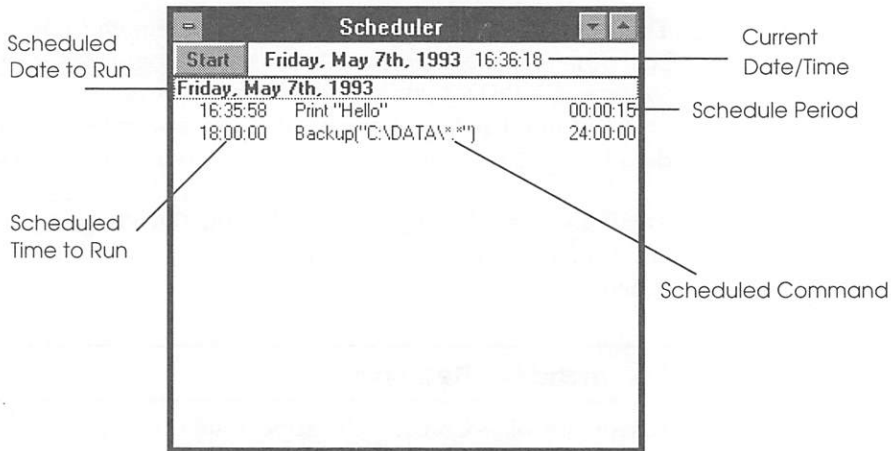
Continued

Command	Description
SchedQData	Returns information about scheduled events.
SchedRemove	Removes an event from the Scheduler.
SchedSet	Adds an event to the Scheduler.

Many of these commands and functions are discussed in this chapter. For more detailed information, refer to the *CA-Realizer Language Reference Guide*.

The Anatomy of the Scheduler

The Scheduler window is comprised of several parts; the diagram below outlines some basic terms associated with the Scheduler:



Events	The Scheduler shows the current date and time and a list of all the scheduled commands, called <i>events</i> . Events appear with the time they are scheduled to run, and if they are periodic, an indication of how often they are executed.
Starting/Stopping the Scheduler	You can start and stop the Scheduler with the START/STOP button. When the Scheduler is stopped, no events are run. When the Scheduler is started, all the events that should have run while it was stopped will be executed immediately, one at a time.
Controlling Events	You can remove events from the Scheduler by selecting them and pressing the DEL key. Likewise, you can remove all the events scheduled for a given day by selecting that date in the Scheduler and pressing DEL. You can also control the Scheduler through a CA-Realizer program.

Invoking the Scheduler

Invoke the Scheduler by either choosing the SHOW SCHEDULER command in the CA-Realizer WINDOW menu or by executing a command from a program. By incorporating the Scheduler, a program can operate in real time. The Scheduler allows programs to take action on their own, without waiting passively for user input.

How Scheduled Commands are Run

The Scheduler only runs an event when there is no other CA-Realizer program running. It will not stop a program to process an event. A modal form, for example, prevents scheduled events from executing.

When CA-Realizer finishes running a program or a scheduled event, it checks the Scheduler to see if the time for the next event has passed. If so, it runs that event immediately. Therefore, scheduled events are never guaranteed to be run at the time they are scheduled, but they will eventually run some time after.

Scheduling an Event

To place an event in the Scheduler, use the SchedSet function:

```
EventID = SchedSet(Command, Start [, Hours  
[, Minutes [, Seconds]]][; _StartAtNext])
```

SchedSet returns an event number for the scheduled event (*EventID*). You can then use this number to remove the event from the Scheduler with SchedRemove.

Command is a string containing a CA-Realizer command (such as BEEP), a procedure call, or a module command (such as RUN "MyFile"). *Start* is a date-time value indicating the first time CA-Realizer should execute the command line.

If you do not specify the interval parameters—*Hours*, *Minutes*, and *Seconds*—CA-Realizer runs the event only once. If you use the interval parameters, CA-Realizer reschedules the event each time the event is run. For example, the following statement causes CA-Realizer to beep every minute and a half:

```
SchedID = SchedSet("BEEP", QDate, 0, 1, 30)
```

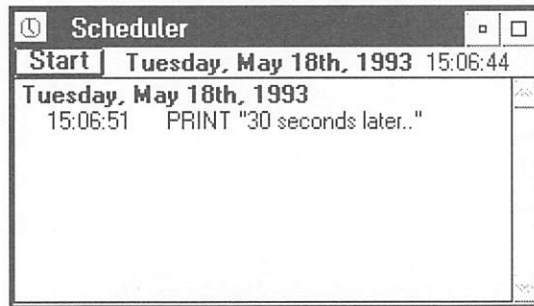
The `_StartAtNext` modifier prevents the first execution of the command if the start time has already passed. Normally, CA-Realizer immediately executes any events that you schedule for a time that has already passed. If the command is scheduled with `_StartAtNext`, the command does not run until the next number of integral intervals from *Start*.

To execute a `PRINT` statement 30 seconds from now, type in and execute the following:

```
Event1 = SchedSet("PRINT ""30 seconds later..""", \\
    AddToTime(QDate, 0, 0, 30))
```

The event defined by the above `SchedSet` function is split between two parameters. The first parameter is the program statement that you want to execute, and the second parameter is the date-time value of the event. In this example, the `AddToTime` function adds 30 seconds to the current time; the value of the second parameter is the result of this computation.

Look at the Scheduler window below to view the two lines of text that describe the event. The first line, written in bold text, is the day, month, and year in which the event occurs. The second line indicates the time at which CA-Realizer should invoke the event and execute the program statement.



To see how CA-Realizer triggers this event, choose the **START** button in the Scheduler window. (You must start the Scheduler before CA-Realizer will execute any events.) Open the Print Log to see the result. When the event time matches the computer's clock time, CA-Realizer executes the **PRINT** statement. While the event is running, a square mark appears next to the event label. CA-Realizer then removes the event from the Scheduler list.

You can schedule an event relative to the current date and time, as in the previous example, or for any specific date-time value. For instance, to add a second event to the Scheduler, type:

```
Event2 = SchedSet("PRINT "Happy Birthday!", "\\  
StrToDate("7/25/93 00:05"))
```

CA-Realizer executes this event at the particular date and time that the `StrToDate` function returns. If this date has already passed, CA-Realizer executes the event immediately.

Once CA-Realizer executes the two events added to the Scheduler, it removes them from the event list. This type of event is useful for actions that occur only once or at irregular intervals.

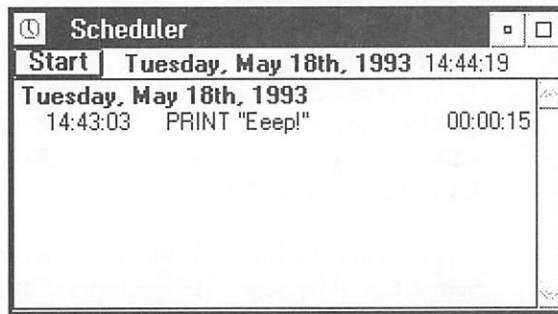
Repeating an Event

You can also create *periodic events*, such as transferring stock data every morning or updating a spreadsheet every 10 minutes. To create a periodic event, add interval parameters to the SchedSet function.

For example, add an event that invokes a PRINT command every 15 seconds:

```
Event1 = SchedSet("PRINT ""Eeep!"" ", \\  
                QDate, 0, 0, 15)
```

Execute this line and look in the Scheduler window below. As before, the event appears in the Scheduler. The second SchedSet parameter indicates the first time CA-Realizer should execute the event. The third through fifth parameters describe the interval, in hours, minutes, and seconds between each repetition. Note that CA-Realizer displays the defined interval on the right side of the event label.



Display the Print Log and watch text accumulate. CA-Realizer executes the PRINT command as soon as you add the event to the Scheduler and every 15 seconds thereafter. *Event1* continues to run at this interval, until the user either stops the Scheduler or removes the event from the Scheduler window.

You can add a second periodic event. This time, the event invokes a procedure called *DisplayTime*. Type in and execute the following program statements, executing them about five seconds after the most recent text appears in the Print Log:

```
PROC DisplayTime
    PRINT USING "The time is D(h4:m1:s1 h5)."; \\
        QDate
END PROC
Event2 = SchedSet("DisplayTime", QDate, 0, 0, 15)
```

This second event also has a 15-second interval. The events do not coincide, however, since the time interval is relative to the starting time supplied by the *QDate* function. The text from *Event2* should appear roughly 5 seconds after the text from *Event1*. You will also note that as CA-Realizer executes each periodic event, it reschedules it in the event list. *Event1* and *Event2* change places each time CA-Realizer triggers an event, so that the events remain listed in chronological order.

Starting and Stopping the Scheduler

You must start the Scheduler before scheduled events will run. To start the Scheduler, choose the **START** button in the Scheduler window or execute the `SchedControl(_Start)` command. In either case, while the Scheduler is active, CA-Realizer replaces the **START** button with a **STOP** button.

To prevent scheduled events from executing, stop the Scheduler. The Scheduler is stopped by choosing the **STOP** button or by running the `SchedControl(_Stop)` command.

Start and stop the Scheduler several times. Every time you restart, CA-Realizer immediately executes the events whose trigger time has passed.

Removing an Event from the Scheduler

Once you add a periodic event to the Scheduler, it repeats the task indefinitely. There are two ways to remove an event from the Scheduler.

- The simplest way to stop a periodic event is to stop the Scheduler completely.
- Another way to stop a periodic event is to remove it from the event list.

Note that stopping the Scheduler stops all other pending events as well. As an alternative, use the event number that the `SchedSet` function returns to execute the `SchedRemove` command.

The `SchedRemove` Command

For example, the following command removes *Event1* from the Scheduler:

```
SchedRemove(Event1)
```

The `SchedRemove` command is typically triggered by a menu selection, or as the result of a conditional statement.

The `SchedRemove` command itself may be scheduled to remove an event at a particular time. For instance, *Event2* will be removed from the Scheduler one minute after CA-Realizer executes the following line:

```
Event4 = SchedSet("SchedRemove(Event2)", \\  
AddToTime(QDate, 0, 1))
```

Using the DEL Key

You can also remove events from the Scheduler by selecting them in the Scheduler window and pressing the DEL key. To remove an entire day of events, select a bold date label and press DEL.

Clearing All Events

To clear all scheduled events at once, use the `SchedControl(_Clear)` command.

Querying the Scheduler and Its Contents

Use the SchedQData command to query Scheduler parameters for a specific event, *EventID*:

```
ReturnValue = SchedQData(EventID)
```

ReturnValue is a family that contains the following members:

Member	Description
<i>cmdID</i>	The command ID.
<i>cmd</i>	The command string.
<i>startDT</i>	The day and time at which the command is to be executed next.
<i>incHour</i>	The hour portion of the interval.
<i>incMin</i>	The minutes portion of the interval.
<i>incSec</i>	The seconds portion of the interval.

Note: If there are no parameters, the query returns the family of arrays for all commands.

Chapter 20

Help Files

In This Chapter	20-1
Using the System Help Manager	20-1
Accessing the Help Manager	20-2
Accessing a Help File	20-2
Using a Help Procedure.....	20-3

Chapter 20

Help Files

In This Chapter

The Windows and OS/2 Help Managers enable applications to present online help information to the user. Typically, this information is displayed when the user presses F1 or selects a command from the HELP menu.

Help can display general information about an application or it can be *context-sensitive*. Context-sensitive help provides information about the current menu item that is selected, the front window, or the current action the user is undertaking. CA-Realizer applications can provide both general information and context-sensitive help.

Using the System Help Manager

The system Help Manager can be used to display help files created with the Help Compiler, which is included with CA-Realizer. These help files contain text that is indexed by topic as well as hypertext links.

CA-Realizer can invoke the Help Manager at any time and tell it to display information about a specific topic in a given help file.

Accessing the Help Manager

Use the `Help` command in your application to run the Help Manager. Its general form is:

```
Help(HelpFile, Topic)
```

The `Help` command tells the Help Manager to load the file *HelpFile*. If a file extension is not specified, `.HLP` is used. If a path is not specified, the `_LoadDir` path is searched.

Topic is the item in the `.HLP` file that should initially be displayed when the help file is loaded. If you specify `_HelpIndex`, the main index of the `.HLP` file appears.

Accessing a Help File

You can access a custom help file from your application's `HELP` menu by hiding the `CA-Realizer HELP` menu with the `MenuControl(_Hide)` command and creating a new one with `MenuNew`. This is demonstrated by the following program:

```
PROC MyHelpMenuProc(params)
  SELECT CASE params[_ItemNum]
  CASE 99
    Help("HelpFile.HLP", _HelpIndex)
  CASE ELSE
    Help("HelpFile.HLP",
      HelpTopics[params[_ItemNum]])
  END SELECT
END PROC

MenuCommands = {"Help on &Stocks", \
  "Help on &Bonds", "Help on &Pricing"}
HelpTopics = {"Stocks", "Bonds", "Pricing"}

MenuSelect(_HelpMenu)
MenuControl(_Hide)

MyHelpMenu = MenuQUnique
MenuNew(MyHelpMenu; "&Help")
MenuSetCmd(Index(3), MenuCommands)
MenuSetCmd(_Separator)
MenuSetCmd(99, "&Index")
MenuSetProc(MyHelpMenuProc)
MenuControl(_Show)
```

For more information on help files, refer to the documentation accompanying the Windows Help Compiler and the OS/2 IPF Compiler.

Using a Help Procedure

You can provide context-sensitive help by adding push buttons to a form that, when pressed, display further information about the form. Alternatively, you can specify a help procedure for a tool or menu with the `HelpSetProc` command. Its general form is:

```
HelpSetProc[(ProcName | ModuleName)]
```

ProcName is the name of a procedure that is invoked when the user presses F1. You can use a module (macro) file instead of a procedure by specifying it as *ModuleName*. `HelpSetProc` without any parameters detaches the current help procedure.

When the user presses F1 in a tool or menu that the program created, the help procedure associated with that tool or menu is invoked. The *params[_ItemType]*, *params[_ItemNum]*, and *params[_FormNum]* parameters in the help procedure parameter array indicate the tool that was active when help was requested. The help procedure can use this information to present help about the application in general, about a certain class of tools, or about a specific tool.

The general structure of a help procedure is as follows:

```
PROC MyHelpProc(params)
  SELECT CASE params[_ItemType]
  CASE _Chart
    'Help on charts
    'Could do SELECT CASE params[_ItemNum]
    '  to give specific help for each chart
  CASE _Form
    'Help on forms
    'Could do SELECT CASE params[_ItemNum]
    '  to give specific help for each form
  CASE _Menu
    'Help on menus
    '  params[_MenuNum] is the menu selected
    '  params[_ItemNum] is the item within the
    '    menu
  END SELECT
END PROC
HelpSetProc(MyHelpProc)
```

If the user presses F1 when the active window is a CA-Realizer program window, a CA-Realizer menu, or the Scheduler, the help procedure is not invoked. The normal CA-Realizer help is used instead.

If the help procedure determines that the user actually wants help about CA-Realizer, it can force the normal help window to appear by setting *params[_UseRealizer]* to 1. When the help procedure is finished, the system Help Manager runs.

Chapter 21

CA-Realizer System Variables

In This Chapter	21-1
SetSys and QSys Parameters	21-2
The Size Variables	21-5
The Path Variables	21-7
The Log Variables	21-9
Default PRINT Formats	21-10
Command-Specific System Variables	21-11
System Constants	21-13

Chapter 21

CA-Realizer System Variables

In This Chapter

CA-Realizer provides the QSys function and the SetSys command for querying and tuning the CA-Realizer system. There are a number of system variables that are accessible *only* through these commands.

SetSys changes the value of a system variable, while QSys returns the current value. While not all parameters can be changed with the SetSys command, they can be queried using the QSys function.

CA-Realizer's system variables contain information you can use to customize the system. For example, one variable indicates where to look when loading data files, while another sets the log where CA-Realizer prints errors. System variables are similar to the DOS environment space.

The general formats for SetSys and QSys are as follows:

```
SetSys(Which, Value1 [, Value2])  
gg = QSys(Which [, Color])
```

This chapter describes each of the valid parameters for QSys and SetSys.

SetSys and QSys Parameters

The following table lists the valid *Which* parameters for both QSys and SetSys. Those listed with an asterisk cannot be changed with SetSys; they can only be queried with QSys.

<i>Which</i>	<i>Value</i>
<code>_AlphaFormat</code>	A string containing the default format string for string values.
<code>_CaseSensitive</code>	A numeric scalar that indicates whether lowercase and uppercase letters should be treated the same in string comparisons. A value of 1 indicates that uppercase and lowercase are distinct. A value of 0 indicates that they are treated the same.
<code>_CenturySplit</code>	A numeric scalar that indicates the dates that are for the 20 th and 21 st centuries. CA-Realizer considers any year that you enter as a two-digit number, larger than this value, as part of the 20 th century, and any year smaller than this value, as part of the 21 st century. For example, if <code>_CenturySplit</code> is 52, then 1/1/52 is the same as 1/1/1952, while 1/1/51 is the same as 1/1/2051.
<code>_CmdLine</code>	A string containing the command line arguments passed to CA-Realizer when it was launched.
<code>_Color</code> *	A three-element vector representing the RGB values for <i>Color</i> . <i>Color</i> can be a CA-Realizer color constant or a vector of three RGB values, in which case CA-Realizer returns the RGB vector that most closely matches the color the system monitor displays. All RGB values are real values from 0 to 1.

Continued

Continued

Which	Value
<code>_DateFormat</code>	A string containing the default format string for date-time values.
<code>_ErrorLog</code>	A two-element array containing the ID of the log into which CA-Realizer writes error messages. The first element is the number of the log. The second element is the number of the form containing the log (or 0 if the log is not within a form).
<code>_FPErr</code>	A numeric scalar that indicates what CA-Realizer does when a floating point error occurs. If it is non-zero, the program stops and CA-Realizer displays an error. If it is 0, CA-Realizer uses an appropriate value for the invalid number.
<code>_HWnd *</code>	The window handle for the CA-Realizer window. You can use this handle with external procedures and functions.
<code>_LoadDir</code>	A string containing a set of paths separated by semicolons. CA-Realizer searches these paths when you open files or load bitmaps in a form.
<code>_MacroDir</code>	A string containing a set of paths separated by semicolons. CA-Realizer searches these paths when you RUN macros or load external DLLs.
<code>_OpSys *</code>	A numeric value indicating the operating system. Possible values include: <code>_Win3</code> (Windows 3.X) and <code>_OS2</code> (OS/2 2.X).
<code>_OrigDir *</code>	A string indicating the directory in which the CA-Realizer executable file is located.
<code>_OutputWidth</code>	A numeric scalar indicating the maximum size, in characters, of an output line in a log. Lines that are longer wrap to the next line.

Continued

Continued

Which	Value
<code>_PrintBest</code>	A numeric scalar indicating whether CA-Realizer should try to use the best font possible when printing on non-PostScript printers. This increases print quality and also increases print time if set to 1.
<code>_PrintLog</code>	A two-element array containing the ID of the log into which CA-Realizer writes printed output. The first element is the number of the log. The second element is the number of the form containing the log (or 0 if the log is not within a form).
<code>_ProgDir *</code>	A string indicating the directory of the CA-Realizer file that is running. Within a procedure or function, this is the directory containing the file that defines the procedure or function.
<code>_RealFormat</code>	A string containing the default format string for real values.
<code>_Separator</code>	A string containing the character used by FileImport and FileExport to separate columns.
<code>_Size</code>	A numeric array representing the size of the CA-Realizer window. If you minimize or maximize the window, the array will have 1 element, either <code>_Minimize</code> or <code>_Maximize</code> . Otherwise, the array will have four elements representing the position and size of the CA-Realizer window (left, top, width, and height) in pixels.
<code>_SizeInside</code>	A four-element numeric array (in pixels) representing the size of the client region of the CA-Realizer window (left, top, width, and height) relative to the screen.

Continued

Continued

Which	Value
<code>_SizeMDI</code>	A four-element numeric array (in pixels) representing the insets of the MDI region within the main window (left, top, right, and bottom).
<code>_Title</code>	The title of the CA-Realizer application window.
<code>_Version *</code>	A string containing the CA-Realizer version number.

The Size Variables

`_Size`

The `_Size` variable can be used with the `SetSys` command and the `QSys` function to specify the size and position of the CA-Realizer main window, or to return its current size and position:

```
SetSys(_Size, Size)
Size = QSys(_Size)
```

The *Size* parameter is an array of one, two, or four real values. As a single element, specify *Size* as `_Minimize` or `_Maximize` to indicate that the CA-Realizer window can be either minimized or maximized, respectively. If *Size* has two elements, they are used as the new upper-left position of the window. If four elements are specified, they represent the left, top, width, and height values for the window. All values are specified using CA-Realizer's position notation.

For example:

```
'Maximize the CA-Realizer window
SetSys(_Size, {_Maximize})

'Make CA-Realizer window 1/4 of the Windows screen
SetSys(_Size, {_Center, _Center, 50 pct, 50 pct})
```

QSys(_Size) returns a one-element array if the window is minimized or maximized. Otherwise, it returns a four-element array representing the left, top, width, and height values in pixels.

_SizeInside

The _SizeInside variable can be used with the SetSys command and the QSys function to specify the inside dimensions of the CA-Realizer main window, or to return its current inside size:

```
SetSys(_SizeInside, Size)
Size = QSys(_SizeInside)
```

For example, run the following and compare it to the second example presented for the _Size variable presented above:

```
'Make the inside of the CA-Realizer window 1/4 of
the Windows screen
SetSys(_SizeInside, {_Center, _Center, 50 pct, 50
pct})
```

The difference between the two is the border and menu bar of the CA-Realizer window.

QSys(_SizeInside) returns a one-element array if the window is minimized or maximized. Otherwise, it returns a four-element array representing the left, top, width, and height values in pixels.

_SizeMDI

You can use the `_SizeMDI` variable to set the size of the inner MDI window used for program windows and tool windows, or to return the current size of the MDI window:

```
SetSys(_SizeMDI, Size)
Size = QSys(_SizeMDI)
```

In this case, *Size* is a four-element array, indicating the left, top, right, and bottom margins, respectively, between the main window and the MDI region.

`QSys(_SizeMDI)` always returns a four-element array with the MDI inset values.

The Path Variables

_LoadDir

The `_LoadDir` variable can be used to specify the list of directories searched when files are opened (for example, with `FileOpen` or `FileImport`) or when bitmaps are loaded into a form with `FormSetObject`. Used with `QSys`, `_LoadDir` returns the current search path.

```
SetSys(_LoadDir, Path)
Path = QSys(_LoadDir)
```

Path is a string containing a set of paths separated by semicolons.

`QSys(_LoadDir)` returns the current search path as a string containing a set of paths separated by semicolons.

You can also use the `AddSys(_LoadDir, NewDir)` command to add the directory names to the specified CA-Realizer system search path variable. For more information about `AddSys`, see the “File I/O” chapter.

_MacroDir

The RUN command searches the _MacroDir path list to load macro files, and the EXTERNAL command searches the _MacroDir path list to load non-resident Dynamic Link Libraries (DLLs).

```
SetSys(_MacroDir, Path)
```

```
Path = QSys(_MacroDir)
```

Path is a string containing a set of paths separated by semicolons.

QSys(_MacroDir) returns the current search path as a string containing a set of paths separated by semicolons.

You can also use the AddSys(_MacroDir, *NewDir*) command to add directory names to the specified CA-Realizer system search path variable. For more information about AddSys, see the "File I/O" chapter.

_ProgDir

To determine the directory in which the currently running program resides, use the QSys function as follows:

```
Path = QSys(_ProgDir)
```

Within a procedure or function, this is the directory containing the file that defines the procedure or function.

QSys(_ProgDir) returns a string containing the directory name. An empty string is returned if the program is in a log or in an untitled program window, or if the QSys command is in an EXECUTE statement.

_ProgDir is commonly used for managing CA-Realizer programs that have a number of data and bitmap files. If you place these files in a subdirectory relative to the program, you can add a path created from _ProgDir to the _LoadDir path.

For instance, a program called DataBase might keep its associated data files in a subdirectory called DBDir located in the same directory as the DataBase.RLZ program. Using this example, the following statement ensures that CA-Realizer finds the data files:

```
SetSys(_LoadDir, QSys(_LoadDir) + ";" + \\
    QSys(_ProgDir) + "DBDir")
```

An easier method, however, is to use the AddSys command as follows:

```
AddSys(_LoadDir, QSys(_ProgDir) + "DBDir")
```

For more information about AddSys, refer to the "File I/O" chapter.

_OrigDir

To determine the directory in which a CA-Realizer executable file is located, use the _OrigDir variable as follows:

```
Path = QSys(_OrigDir)
```

QSys(_OrigDir) returns a string containing the directory name.

The Log Variables

_PrintLog and
_ErrorLog

The output of the PRINT command, if no log is specified, goes to the Print Log. Likewise, all error messages and warnings go to the Error Log. You can change these defaults by specifying a different log for the _PrintLog and _ErrorLog system variables. _PrintLog and _ErrorLog can also be used with the QSys function to determine the current Print or Error log.

The general syntax for each is as follows:

```
SetSys(_PrintLog, LogNum [, FormNum])
```

```
LogNum = QSys(_PrintLog)
```

```
SetSys(_ErrorLog, LogNum [, FormNum])
```

```
LogNum = QSys(_ErrorLog)
```

SetSys(_PrintLog) and SetSys(_ErrorLog) can take a log number (*LogNum*), both a log number and a form number (*FormNum*), or a two-element array containing both.

If the log receiving output is closed, output will again be sent to the original log. You can use SetSys(_PrintLog, _PrintLog) and SetSys(_ErrorLog, _ErrorLog) to return the logs to their original states.

QSys(_PrintLog) and QSys(_ErrorLog) both return a two-element array with the log number and the form number (0 if the log is not in a form).

Default PRINT Formats

The following system variables are the default format strings used by PRINT, the spreadsheet tool, and other commands for displaying data.

AlphaFormat

Use the AlphaFormat variable with the SetSys command or QSys function to specify the default format string for string values or to determine the current default format, respectively:

```
SetSys(_AlphaFormat, Format)
```

```
Format = QSys(_AlphaFormat)
```

Format is a string containing the default format string for string values. The default is &.

RealFormat

You can use RealFormat with the SetSys command and QSys function to indicate the default format string for real values, or to determine the current default format, respectively:

```
SetSys(_RealFormat, Format)
```

```
Format = QSys(_RealFormat)
```

Format is a string containing the default format string for real values. The default is #####.####.

_DateFormat To specify the default format string for date-time values, or to determined the current default format, use the `_DateFormat` variable with the `SetSys` command or `QSys` function as follows:

```
SetSys(_DateFormat, Format)
Format = QSys(_DateFormat)
```

Format is a string containing the default format string for date-time values. The default is `D(M1\D1\Y1)`.

_OutputWidth The `_OutputWidth` system variable is a numeric scalar used to specify the number of characters to be printed to a log before the text is wrapped to the next line. Used with the `QSys` function, `_OutputWidth` returns the currently specified width.

```
SetSys(_OutputWidth, Width)
Width = QSys(_OutputWidth)
```

Width must be between 10 and 512—the default is 512. You can change this value to prevent text from being displayed beyond the right edge of a log.

Note: For more information about the usage of each of these parameters, refer to “Controlling Input and Output.”

Command-Specific System Variables

_Separator The `_Separator` system variable, which is discussed in greater detail in “Controlling Input and Output,” is used to specify the character to be used to separate fields in files manipulated by `FileImport` and `FileExport`. Used with the `QSys` function, `_Separator` is used to determine the currently specified character.

```
SetSys(_Separator, SepChar)
SepChar = QSys(_Separator)
```

SepChar is the character used to separate fields—the default is a tab (ASCII 9).

_CenturySplit

To specify the last year to be automatically considered as part of the 20th century when reading date formats with a two-digit year (as with StrToDate or FileImport), use the _CenturySplit variable. To determine the current value of _CenturySplit, use the QSys function:

```
SetSys(_CenturySplit, Split)
Split = QSys(_CenturySplit)
```

Split is a real value from 0 to 99—the default is 50.

If _CenturySplit were 62, for example, any year from 62 to 99 would be considered part of the 20th century (for example, 5/16/62 is the same as 5/16/1962), while years from 0 to 61 would be part of the 21st century (for example, 7/14/61 would be the same as 7/14/2061).

_CaseSensitive

To specify whether uppercase and lowercase letters should be considered distinct and separate or the same when comparing strings and in the InStr, SubStr\$, and StatSort functions, use the _CaseSensitive variable. Used with the QSys function, _CaseSensitive returns the current setting.

```
SetSys(_CaseSensitive, Sensitive)
Sensitive = QSys(_CaseSensitive)
```

Sensitive should be 1 when you want uppercase and lowercase letters to be considered distinct from one another. Set _CaseSensitive to 0 when uppercase and lowercase should be treated as the same. The default is 1.

For more information about _CaseSensitive, see the “Manipulating Strings” chapter.

_FPError

To specify the action to be taken by CA-Realizer when a floating point error occurs, or to determine the current value of `_FPError`, use the `_FPError` variable with the `SetSys` command and `QSys` function as follows:

```
SetSys(_FPError, Error)
Error = QSys(_FPError)
```

`Error` is a numeric scalar. If it is non-zero, the program stops and CA-Realizer displays an error. If it is 0, CA-Realizer uses an appropriate value for the invalid number. The default value of `_FPError` is 0.

For more information about `_FPError`, see the “Manipulating Numbers” chapter.

System Constants

_Color

Use the `_Color` variable with the `QSys` function to return a vector of three real values from 0 to 1, representing the red, green, and blue values for the given color constant:

```
RGB = QSys(_Color; Color)
RGB = QSys(_Color; Red, Green, Blue)
```

For example:

```
c = QSys(_Color; _Orange)
ROWPRINT c
1.0000      0.5020      0.0000
```

CA-Realizer rounds RGB values when they are used on the screen. If you use separate red, green, and blue values with `QSys(_Color)`, the returned values will be the actual rounded values that CA-Realizer uses.

_Version

You can use the `_Version` variable to return a string indicating the version of CA-Realizer that is running as follows:

```
Version = QSys(_Version)
```

<code>_OpSys</code>	To determine the operating system, use the <code>_OpSys</code> variable with the <code>QSys</code> function as follows: <pre><i>OpSys</i> = QSys(<i>_OpSys</i>)</pre> This returns either <code>_Win3</code> or <code>_OS2</code>.
<code>_Hwnd</code>	<code>QSys(_Hwnd)</code> returns the Windows handle for the main CA-Realizer window—this can be used in EXTERNAL procedures and functions. <pre><i>Hwnd</i> = QSys(<i>_Hwnd</i>)</pre>
<code>_CmdLine</code>	Use the <code>_CmdLine</code> variable to specify the command line arguments to be passed to CA-Realizer when it is launched, or to determine what arguments were passed. <pre>SetSys(<i>_CmdLine</i>, <i>CmdLine</i>)</pre> <pre><i>CmdLine</i> = QSys(<i>_CmdLine</i>)</pre> <i>CmdLine</i> is a string containing the command line arguments.
<code>_PrintBest</code>	To force the printer to try to find a better match for fonts and graphics, or to determine the current setting of <code>_PrintBest</code>, use the <code>_PrintBest</code> variable with the <code>SetSys</code> command and <code>QSys</code> function as follows: <pre>SetSys(<i>_PrintBest</i>, <i>PrintBest</i>)</pre> <pre><i>PrintBest</i> = QSys(<i>_PrintBest</i>)</pre> If <i>PrintBest</i> is set to 1, the printer will try to find a better match. Note that setting <code>_PrintBest</code> to 1 will cause print time to be increased. The default is 1.
<code>_Title</code>	<code>_Title</code> can be used to set or query the title of the CA-Realizer application window as follows: <pre>SetSys(<i>_Title</i>, <i>Title</i>)</pre> <pre><i>Title</i> = QSys(<i>_Title</i>)</pre> <i>Title</i> is the name of the CA-Realizer application window. The default is CA-Realizer.

Chapter 22

Accessing Operating System Commands

In This Chapter	22-1
Launching Other Applications	22-1
Running Operating System Commands	22-3
Creating Batch Files with CA-Realizer	22-4
How Windows Commands Synchronize with CA-Realizer	22-5
Setting the System Date and Time	22-6

Chapter 22

Accessing Operating System Commands

In This Chapter

Windows and OS/2 are multitasking environments that enable the user to run several applications at once. As a part of these environments, CA-Realizer programs can launch other applications and run operating system commands.

In this chapter you will learn how to:

- Launch other applications
- Run operating system commands
- Create batch files
- Synchronize Windows and CA-Realizer commands

Launching Other Applications

CA-Realizer runs in the Windows and OS/2 multitasking environments and is capable of launching other applications.

You can use the SHELL command to start applications:

```
SHELL OSCommand [, AppStyle]
```

OSCommand is a string containing the name of the command to run, along with any command line arguments. You must specify the file extension for .BAT and .COM files, but you can omit the extension for .EXE and .PIF files.

For example, to run CA-Compete! and load the spreadsheet TEST.MDB, use the following statement:

```
SHELL "COMPETE TEST.MDB"
```

This makes CA-Compete! the active application.

AppStyle specifies the size of the application window and can be any one of the following constant values:

<i>AppStyle</i>	Description
<code>_Normal</code>	The new application appears with its default size.
<code>_Minimize</code>	The new application is minimized when it appears.
<code>_Maximize</code>	The new application is maximized when it appears.
<code>_Inactive</code>	The new application appears minimized, but is not the foremost.
<code>_DOS</code>	The command is run in a DOS window.

For example, you can load CA-Compete! with the spreadsheet TEST.MDB, but then you can force CA-Compete! to remain inactive, making it ready to participate in Dynamic Data Exchange (DDE):

```
SHELL "COMPETE TEST.MDB", _Inactive
```

Non-graphical programs and commands are run in a DOS window.



Under Windows, you can prevent the flicker associated with switching to the DOS screen by creating a .PIF file for the DOS application. This indicates that it can be run in a "windowed" DOS screen. If the background execution flag is set in the .PIF file, multiple DOS sessions can be run at once.

Running Operating System Commands

CA-Realizer provides an environment in which you can run operating system commands as demonstrated by the syntax below:

```
SHELL DOSCommand, _DOS
```

OS/2 and DOS commands such as COPY can be specified as follows:

```
SHELL "COPY C:\*.RLZ A:", _DOS
```

If CA-Realizer needs the results of the command, the output can be redirected to a file:

```
SHELL "DIR > TEMP.TMP", _DOS
```

Once the shelled command has finished, you can use FileRead or FileImport to read and manipulate the temporary file. CA-Realizer may or may not wait for the command to finish, depending on the COMMAND.PIF file.

You can use the IDLE command to force CA-Realizer to wait a certain amount of time to allow the file to be created. Although the program continues to run, CA-Realizer is more receptive to other applications that are running simultaneously. (For more information, see the IDLE command in the *CA-Realizer Language Reference Guide*.)

For example:

```
SHELL "DIR > TEMP.TMP", _DOS
LOOP
    'Wait five seconds
    IDLE 5

    'Check for the file
    IF FileQ("TEMP.TMP", _Exists) THEN
        EXIT LOOP
    END IF
END LOOP
```

A DOS window in which the user can type commands can be created using the statement below:

```
SHELL "COMMAND.COM"
```

When the command window is no longer needed, type EXIT at the command line prompt to close it.

Creating Batch Files with CA-Realizer

The CA-Realizer file commands can create a batch file that can be executed with the SHELL command as shown below:

```
INPUT "Directory to backup:"; SourceDir
FileName = "TEMP.BAT"
FileNum = FileQUnique

FileOpen(FileNum, FileName, _Write)
FileWrite(FileNum, "CD " + SourceDir)

'If a drive was specified, have the batch file
'add a change drive command (i.e., "C:")
IF Mid$(SourceDir, 2, 1) = ":" THEN
    FileWrite(FileNum, Left$(SourceDir, 2))
END IF

FileWrite(FileNum, "MKDIR BACKUP")
FileWrite(FileNum, "COPY *.* BACKUP")
FileClose(FileNum)

SHELL FileName
```

This program prompts the user for a directory and creates a batch file to back up that directory. The batch file first sets the working drive and directory, and then makes a subdirectory called BACKUP. All the files in the source directory are then copied into BACKUP.

Eventually the batch file TEMP.BAT can be deleted with FileDelete or an equivalent DOS command. However, you must design the program so that the DOS shell is finished with the file. The problems of synchronization are discussed in the next section.

How Windows Commands Synchronize with CA-Realizer



Because Windows is a multi-application environment, any time a program is idle, another program may get the chance to run. When CA-Realizer launches another program with the SHELL command, it tells Windows to launch the application and then continues with what it was doing. CA-Realizer does not wait for the shelled command to complete. In fact, CA-Realizer cannot tell when the other command has finished.

This can cause synchronization problems for programs that need to execute a DOS command or program and wait for its completion. Depending on the COMMAND.PIF file, the other command may get all or a little of the processing time, or it may be completely preempted until the CA-Realizer program stops.

CA-Realizer does not receive any feedback from a simple DOS command or program if an error occurs. You should explicitly check for files that were supposed to be created with FileQ(_Exists), or you should verify the time stamp of a file with FileQ(_DateTime) to prove that it was updated.

One possible technique, is to use a file as a flag to indicate that a batch file is complete. CA-Realizer can delete this flag file before launching the batch file, and the batch file can create the flag file when it is finished. CA-Realizer can then loop and check for the existence of this file to know when the batch file has finished.

You can use the IDLE command to force CA-Realizer to yield processing time to the batch file as shown below:

```
FileDelete("FLAGFILE.TMP")
SHELL "BATCH.BAT"
LOOP
    'Wait five seconds
    IDLE 5

    'Check for the file
    IF FileQ("FLAGFILE.TMP", _Exists) THEN
        EXIT LOOP
    END IF
END LOOP
```

Setting the System Date and Time

Occasionally, an application requires that you temporarily change the environment date or time. Use `SetSystemDate` and `SetSystemTime` for this purpose:

```
SetSystemDate(Date)
```

```
SetSystemTime(Time)
```

These commands change the internal system date and time to the date or time that *Date* and *Time* specifies. *Date* and *Time* are date-time variables.

```
'Update the system date  
today = StrToDate("4/8/91")  
SetSystemDate(today)
```

```
'Update the system clock  
now = StrToDate("3:15 pm")  
SetSystemTime(now)
```

Chapter 23

The Clipboard

In This Chapter	23-1
Manipulating Text	23-1
Copying Text to the Clipboard	23-1
Retrieving Text from the Clipboard	23-2
Manipulating Pictures	23-2
Placing a Picture on the Clipboard	23-2
Retrieving a Picture from the Clipboard	23-2

Chapter 23

The Clipboard

In This Chapter

The Clipboard, which is a buffer maintained by the operating system, is accessible to all its applications and is useful for transferring data from one application to another. CA-Realizer programs can read and write text and picture data to and from the Clipboard, making it accessible to other applications. Users can also utilize the CUT, COPY, and PASTE commands on the CA-Realizer EDIT menu.

Manipulating Text

You can use the CA-Realizer Clipboard functions, ClipSet and ClipGet, to place text on the Clipboard or retrieve text from the Clipboard, respectively.

Copying Text to the Clipboard

You can use the ClipSet function to copy text to the Clipboard as follows:

```
Success = ClipSet(Text)
```

Because another application can lock the Clipboard, it is not always possible to copy text to the Clipboard. If the copy is successful, *Success* is 1; otherwise, *Success* is 0.

Retrieving Text from the Clipboard

To retrieve text from the Clipboard, use the ClipGet function:

```
Text = ClipGet
```

If there is no text in the Clipboard, an empty string is returned.

Manipulating Pictures

To place pictures on the Clipboard or to retrieve pictures from the Clipboard, use the PictureClipSet and PictureClipGet functions, respectively.

Placing a Picture on the Clipboard

You can place a picture on the Clipboard, using the PictureClipSet function as follows:

```
PicNum = PictureClipSet([PicNum])
```

This function copies the current or specified picture to the Clipboard. If the copy is successful, PictureClipSet returns 1; otherwise, it returns 0.

Retrieving a Picture from the Clipboard

Use the PictureClipGet function to retrieve a picture from the Clipboard as follows:

```
PicNum = PictureClipGet([Format])
```

This function retrieves a bitmap or a metafile from the Clipboard and returns a new picture number. *Format* can be `_Bitmap` or `_Metafile`. If omitted, a bitmap is retrieved.

If there is no picture in the Clipboard, *PicNum* is 0.

Chapter 24

Serial Communications

In This Chapter	24-1
Opening a Serial Port	24-2
Reading from a Serial Port	24-3
Writing to a Serial Port	24-4
Querying a Serial Port	24-5
Closing a Serial Port	24-6

Chapter 24

Serial Communications

In This Chapter

This chapter discusses CA-Realizer support for direct access to the serial communications ports. A port can be opened in any configuration, and then used for bi-directional reading and writing. The following topics are discussed:

- Opening a serial port
- Reading from a serial port
- Writing to a serial port
- Querying a serial port
- Closing a serial port

CA-Realizer provides five standard commands and functions that make it easy to access data through the serial communications ports of your computer as follows:

Function	Description
CommOpen	Opens a serial port for reading and writing.
CommClose	Closes a serial port.
CommRead	Reads data from a serial port.
CommWrite	Writes data to a serial port.
CommQ	Returns information about a serial port.

Opening a Serial Port

A serial communications port can be opened in any configuration. The `CommOpen` command opens a communications port and prepares it for data transfer:

```
CommOpen(PortNum, BaudRate, ReadBuffSize,  
         WriteBuffSize, ByteSize, StopBits, Parity)
```

When used to open a serial port, `CommOpen` creates an input buffer and output buffer to hold the number of bytes being read from—or transmitted to—another device. This is because at high transmission rates, characters can arrive or be sent faster than computers can process them.

PortNum is the number of the port (1, 2, 3, or 4); *BaudRate* is the transmission baud rate; *ReadBuffSize* is the size of the buffer used to hold incoming data (it should be large enough to accommodate the rate at which data will be received and processed); *WriteBuffSize* is the size of the buffer used to hold outgoing data (it should be large enough to accommodate the largest block of data that will be sent at once); and *ByteSize* is the size of the data byte (either 7 or 8).

The number of stop bits to use—either 1, 1.5, or 2—is specified by *StopBits*. Additionally, *Parity* can be one of the following:

<i>Parity</i>	<i>Parity Type</i>
<code>_NoParity</code>	No parity
<code>_OddParity</code>	Odd parity
<code>_EvenParity</code>	Even parity
<code>_MarkParity</code>	Mark parity
<code>_SpaceParity</code>	Space parity

For example, the following statement opens port 2 for a 1200 baud transfer with an 8K read buffer, a 16K write buffer, an 8-bit byte, 1 stop bit, and odd parity:

```
CommOpen(2, 1200, 8192, 16384, 8, 1, _OddParity)
```

Reading from a Serial Port

CommRead

Once a serial communications port is opened, it can be used for bi-directional reading using the CommRead function as follows:

```
Data = CommRead(PortNum, NumBytes)
```

NumBytes is the number of bytes to be read from the communications port, as specified by *PortNum*. The data is returned as a string.

If there are less than *NumBytes* in the buffer of the port, only the data that is currently buffered will be read; CommRead will not wait for the rest of the data to arrive.

CommQ

The number of bytes waiting to be read can be obtained with the CommQ function as follows:

```
Info = CommQ(PortNum)
```

Len

You can determine the actual number of bytes read by using the Len function as follows:

```
ng = Len(Value)
```

The maximum number of bytes that can be read at one time is equal to the port's read buffer size that was specified when the port was opened with CommOpen.

The following example polls the communications port and prints all data received:

```
CommOpen(2, 1200, 2000, 0, 8, 1, _OddParity)
LOOP
  IF CommQ(2)[_CQ_ReadPending] THEN
    PRINT CommRead(2, 80)
  END IF
END LOOP
```

Writing to a Serial Port

CommWrite

Once a serial communications port is opened, it can be used for sending data as follows:

```
CommWrite(PortNum, Data)
```

The CommWrite command writes the data specified in *Data* to communications port number *PortNum*. There must be enough room in the port's write buffer to hold the incoming data; otherwise, some of the data may be lost.

CommQ

You can use CommQ to find the number of bytes waiting to be transmitted as follows:

```
Info = CommQ(PortNum)
```

For example:

```
CommOpen(2, 300, 0, 1024, 8, 1, _OddParity)
LOOP
  'Get a line of data
  data = GetData

  'Wait until there is enough room to send it
  WHILE CommQ(2)[_CQ_WritePending] > Len(data)
    IDLE 1
  END WHILE

  'Send the data
  CommWrite(2, data)
END LOOP
```


Querying a Serial Port

Once a serial communications port is opened, it can be queried using the `CommQ` function:

```
Info = CommQ(PortNum)
```

`CommQ` returns an array of information about *PortNum* to the *Info* variable. The element of the *Info* array can be indexed with any of the following constants:

Constant	Description
<code>_CQ_BaudRate</code>	Baud rate.
<code>_CQ_ReadBufferSize</code>	Input data buffer size.
<code>_CQ_WriteBufferSize</code>	Output data buffer size.
<code>_CQ_ByteSize</code>	Byte size (7 or 8).
<code>_CQ_StopBits</code>	Number of stop bits (1, 1.5, or 2).
<code>_CQ_Parity</code>	Parity type: <code>_NoParity</code> No parity <code>_OddParity</code> Odd parity <code>_EvenParity</code> Even parity <code>_MarkParity</code> Mark parity <code>_SpaceParity</code> Space parity
<code>_CQ_ReadPending</code>	Number of bytes waiting to be read.
<code>_CQ_WritePending</code>	Number of bytes waiting to be transmitted.
<code>_CQ_XON</code>	Status of the data transfer. If XON is 1, data is flowing normally. If XON is 0, the transfer has been temporarily suspended, most likely because of a full buffer.
<code>_CQ_CID</code>	The system port ID. This can be used with EXTERNAL functions.

Here is an example of how CommQ can be used to poll the status of the port:

```
'Slow data transfer with a small write buffer
CommOpen(2, 300, 0, 30, 8, 1, _OddParity)
FOR i = 1 TO 200
    'Get a line of data
    data = GetData

    'Send it through the comm port
    CommWrite(2, data)

    'Wait until it has finished sending
    WHILE CommQ(2)[_CQ_WritePending]
        IDLE 1
    END WHILE
NEXT i
```

Closing a Serial Port

CommClose

Once the program has finished using the port, it should be closed. To close the port, use the CommClose command as follows:

```
CommClose(PortNum)
```

PortNum is the number of the communications port to be closed.

Any data waiting to be sent will be transmitted before the port is closed. For example:

```
CommOpen(2, 1200, 2000, 5000, 8, 1, _OddParity)
CommWrite(2, MyData)
CommClose(2)
```

Chapter 25

Dynamic Data Exchange and CA-RET

In This Chapter	25-1
What Is DDE?	25-1
How DDE Works	25-2
Initiating a Client DDE Session	25-3
Selecting a DDE Session	25-5
DDE Data Formats	25-5
Windows Data Formats	25-6
OS/2 Data Formats	25-7
Sending Messages to a Server	25-8
Responding to Messages from a Server	25-9
Waiting for a DDE Message	25-10
Ending a DDE Session	25-11
A Sample Client Session	25-11
Using CA-Realizer as a DDE Server	25-13
Responding to Messages from a Client	25-16
Obtaining Information about a DDE Session	25-17
Using CA-RET with CA-REALIZER	25-18
Initiating a Conversation with CA-RET	25-18
Executing a CA-RET Command	25-18
Requesting Information from CA-RET	25-19
Terminating the CA-RET Conversation	25-19
A Sample CA-RET Conversation	25-19

Chapter 25

Dynamic Data Exchange and CA-RET

In This Chapter

Windows and OS/2 are multi-application environments that allow the exchange of data between applications through a protocol called *Dynamic Data Exchange* (DDE). CA-Realizer supports this protocol as both a sender and a receiver of data. CA-Realizer also supports a higher level interface to CA-RET, the Computer Associates WYSIWYG report writer.

Both Dynamic Data Exchange and CA-RET are discussed in this chapter. For additional information, refer to the *CA-Realizer User Guide* and your CA-RET documentation.

Note that all references to non-CA-Realizer functions are presented in *bold italic*.

What Is DDE?

The Dynamic Data Exchange protocol allows applications to communicate with each other. In the simplest scenario, one application sends unsolicited information to another. In a more complex situation, applications notify each other whenever a specific piece of information changes. For example, a CA-Realizer application could ask a CA-Compete! application to let it know whenever a particular value has changed. Or, CA-Compete! could ask a CA-Realizer program to keep it posted any time a certain data value is updated.

CA-Realizer programs can initiate a discussion, called a *DDE session*, with another application, or they can respond to other applications' requests for sessions. An application that requests a conversation is called the *client*. An application that responds to a request for a conversation is called the *server*. In general, information flows from the server to the client. The client asks for information, and the server responds by sending data back to the client. An application can conduct both a client and a server session with another application at the same time.

How DDE Works

DDE is a message-based protocol for transferring data between applications. The client application broadcasts a request to begin a discussion with a particular application about a particular topic. All running applications receive this message and must check to see if the request is for them. If they have information about the desired topic, they send a response to the client. If the client acknowledges this response, the other application is established as a server.

Once a client-server relationship has been established for a given topic, the client application can send messages to the server. A message can be a query for a specific piece of data or a request that the client be notified whenever the server has any new information on the topic. The server responds by sending the requested data or by adding the client to a list of applications to be kept informed.

Clients can also send unsolicited information to servers, as well as a set of commands for the server to execute. There is no predefined set of commands or topics—each application defines the set of commands that it will execute and the set of topics that it will discuss. The documentation for the server application should describe these topics.

There are several different formats for DDE data. Though most applications send data as text, there are also formats for sending metafiles, bitmaps, spreadsheet information, and other various types of data. CA-Realizer can handle any of these formats. For more information, see DDE Data Formats later in this chapter.

DDE is a complex topic. For additional information, refer to your Windows and OS/2 documentation, as well as the documentation for those applications (such as CA-Compete!, Microsoft Word for Windows, and Microsoft Excel) that handle DDE.

Initiating a Client DDE Session

To initiate a DDE conversation, the client must provide a standard set of specifications. Every DDE conversation is uniquely identified by an *application name* and a *topic*.

The *application name* is normally the name of the server in the DDE conversation. For example, you can converse with another CA-Realizer application or another GUI application, like CA-Compete!.

A *topic* is a general classification of data within which data items may be exchanged during the DDE conversation. It describes something in the server you want to access, like a spreadsheet file.

DDENew

To initiate a DDE conversation in CA-Realizer, use the DDENew function as follows:

```
ns = DDENew(SessionNum, App, Topic  
            [, Title [, _AllFormats]])
```

SessionNum is an identification number from 1 and 32767 that refers to the session. If *SessionNum* is 0, DDENew automatically selects a unique session number.

Note: The DDEUnique command can also be used to select a unique session number.

App is the name of the application for which the session request is intended, and *Topic* is the name of the topic to discuss. You can name a session by specifying *Title*.

By default, the session receives only data that is sent as text. If you specify the `_AllFormats` modifier, the session receives data sent to it in any format.

If another application is interested in discussing the topic, `DDENew` returns *SessionNum*. If the session could not be established, the return value is 0. CA-Realizer programs should always check this return value before attempting to continue with the data exchange.

If a DDE session with an identification number of *SessionNum* already exists, CA-Realizer closes it before the new one is created.

As an example, consider a DDE session with CA-Compete!. CA-Compete! acts as a server, using the name of the spreadsheet as the session topic:

```
Client = DDEQUnique
success = DDENew(Client, "Compete", "TEST.MDS")
```

This example broadcasts a DDE request for a CA-Compete! application that has information about TEST.MDS. If CA-Compete! is not currently running, or if none of the active CA-Compete! applications currently have a spreadsheet named TEST, the `DDENew` command will fail, returning 0. Since `_AllFormats` was not specified, all communication is with text data.

You can launch CA-Compete! with the TEST spreadsheet before CA-Realizer attempts to establish a DDE connection by using the SHELL command as follows:

```
SHELL "COMPETE TEST.MDS", _Inactive
```

For more information, see *Launching Other Applications in the "Accessing Operating System Commands" chapter*.

Selecting a DDE Session

When a DDE session is created, it becomes the current session and all subsequent DDE commands apply to it.

DDESelect

To select a DDE session, use the DDESelect command as follows:

```
DDESelect(SessionNum)
```

Note that you can also use the DDESelect command to change the current session to *SessionNum*.

DDEQ(_Selected)

The DDEQ(_Selected) function returns the number of the current session.

DDE Data Formats

Data can be sent and received in many formats. The simplest format is text, which can include binary data packed into strings.

As previously stated, unless you specify the `_AllFormats` modifier with `DDENew`, CA-Realizer ignores any messages in a format other than text.

If you specify `_AllFormats`, all messages come through, and their format is indicated by the value *Format*. This value is also used when data is sent to another application in a format other than text. The format numbers are defined by Windows or OS/2.

Windows Data Formats



Under Windows, the formats are listed in the `WINDOWS.H` file. Common Windows data formats, their format numbers, and associated constants are listed below:

Value	Format	Description
1	CF_TEXT	Text.
2	CF_BITMAP	A handle to a bitmap.
3	CF_METAFILEPICT	Metafile picture format.
4	CF_SYLK	Microsoft Symbolic Link format.
5	CF_DIF	Software Arts' Data Interchange Format.
6	CF_TIFF	Tag Image File Format (TIFF).
7	CF_OEMTEXT	Text containing characters in the OEM character set.
8	CF_DIB	Device Independent Bitmap data structure and bits.
9	CF_PALETTE	A handle to a color palette.

Applications can also register their own formats with the Windows *RegisterClipboardFormat* function. For more information about a particular format being used by either the client or server application, refer to that application's documentation, as well as your Windows documentation.

OS/2 Data Formats



Under OS/2, the formats are listed in the PMWIN.H file. Common OS/2 data formats and associated constants are listed below:

Format	Description
SZFMT_BITMAP	Bitmap.
SZFMT_CPTEXT	Text defined by a CPTEXT structure.
SZFMT_DIF	Data Image Format (DIF).
SZFMT_DSPBITMAP	Bitmap representation of a private data format.
SZFMT_DSPMETAFILE	Metafile representation of a private data format.
SZFMT_DSPMETAFILEPICT	Metafile picture representation of a private data format.
SZFMT_DSPTEXT	Text representation of a private data format.
SZFMT_LINK	Link-file format.
SZFMT_METAFILE	Metafile.
SZFMT_METAFILEPICT	Metafile picture defined by an MFP structure.
SZFMT_OEMTEXT	OEM text.
SZFMT_PALETTE	Palette.
SZFMT_SYLK	Synchronous Link format.
SZFMT_TEXT	Null-terminated text string.
SZFMT_TIFF	Tag Image File Format (TIFF).

Applications can also register their own formats with the OS/2 *WinAddAtom* function. For more information about a particular format being used by either the client or server application, refer to that application's documentation, as well as your OS/2 documentation.

Sending Messages to a Server

DDE protocol allows data to be exchanged between the client and server applications in an active DDE conversation. Using CA-Realizer as a client, you can send data to the server using the DDEControl command.

Once a client application has found another application that will act as a server for a topic, it can send messages to request specific information, to request that it be advised of any new information on the topic, to ask the server to execute commands, or to send unsolicited information to the server.

Note: All the data for the messages is text unless the format is explicitly specified with *Format*. Non-text formats can only be used if the session was initiated with *_AllFormats*.

DDEControl(_Request) DDEControl(_Request) sends a message to the server asking for information about *Topic*:

```
DDEControl(_Request, Topic [; Format])
```

If the server application has the requested information, it responds with a *_Data* message, which can be processed by the DDE procedure of the session (this is discussed in greater detail later in this chapter).

DDEControl(_Advise) DDEControl(_Advise) sends a message to the server requesting that the client be sent a *_Data* message any time there is new information on *Topic*:

```
DDEControl(_Advise, Topic [; Format])
```

DDEControl(_Unadvise) DDEControl(_Unadvise) sends a message to the server requesting that it stop advising on *Topic*:

```
DDEControl(_Unadvise, Topic)
```

DDEControl(_Poke) DDEControl(_Poke) sends unsolicited information to the server about *Topic*:

```
DDEControl(_Poke, Topic, Data [; Format])
```

DDEControl(_Execute) DDEControl(_Execute) is used by the client to request that the server execute a series of commands:

```
DDEControl(_Execute, Commands)
```

Commands is the list of commands as defined by the server application.

Responding to Messages from a Server

When the server sends messages back to the client, a DDE procedure should be used to process the messages. To do this, use the DDESetProc command as follows:

```
DDESetProc[(ProcName | ModuleName)]
```

The DDESetProc command associates a DDE message-handling procedure with a DDE session. You must declare this procedure with four parameters:

```
PROC DDEHandler(SessionNum, Message, Topic, Data)
```

If you used the _AllFormats modifier to initiate the DDE session, the DDE message-handling procedure must have a *Format* modifier:

```
PROC DDEHandler(SessionNum, Message, Topic, \\  
Data; Format)
```

SessionNum is the DDE session number and *Message* indicates the type of message. The topic that the message refers to is *Topic*. *Data* is the information from the message.

When CA-Realizer receives a DDE message, the procedure attached to the appropriate session is invoked.

The server normally only sends two types of messages to the client:

- A `_Data` message that comes in response to a `_Request` or `_Advise` message that the client sent earlier
- A `_Close` message that notifies the client that the server is closing down

The general structure of a client DDE procedure is as follows:

```
PROC ClientProc(Session, Message, Topic, Data)
  SELECT CASE Message
    CASE _Data
      DDESelect(Session)
      ' The server has responded to a Request
      ' or Advise. Look at Topic and Data to
      ' decide what to do
    CASE _Close
      ' The server is shutting down
      ' Clean up as necessary
  END SELECT
END PROC
```

Waiting for a DDE Message

Normally, DDE messages come in asynchronously and invoke the DDE procedure when no other part of the program is running. If the program is in a modal state, you can check for DDE messages with the `DDEWait` command as follows:

```
ReturnValue = DDEWait(TimeOut)
```

This command checks the DDE queue for the current session, and if there are any waiting incoming messages, calls the associated DDE procedure.

TimeOut indicates the number of milliseconds to wait for a message to arrive. If the message is successfully processed, the return value is 1; otherwise, it is 0.

Ending a DDE Session

When the client no longer wants to participate in a DDE conversation, it can notify the server with the `DDEControl(_Close)` command. This command closes the current session and frees the associated system resources.

A Sample Client Session

The following program demonstrates DDE with CA-Compete!. The program assumes that a CA-Compete! spreadsheet named *Stocks* exists, and that it contains information about several stocks, including Acme. The STOCKS.MDL sheet can be found in the CA-Realizer MANUAL\PG_CH25 directory, and should be copied into the main CA-Compete! directory.

	High	Low	Last	Volume
Acme	30	20	25	
Bade				
Crell				
Efna				
Gadt				
Howd				
Jagg				

The first row contains information about the Acme stock, including the highest price of the day, the lowest price of the day, and the latest (last) price of the stock. The CA-Realizer DDE client keeps the high and low prices current if the last price changes.

The DDE protocol of CA-Compete! refers to each spreadsheet cell as a topic in the form *Dim.Row.Col*, where *Dim* is the dimension, *Row* is the row, and *Col* is the column. The client application can request the value of a specific cell, ask to be advised of any changes to a cell, or use a `_Poke` message to change the value of a cell:

```
' Manual\PG_25\DDECOMPT.RLZ

PROC ClientProc(Session, Message, Topic, Data)
  SELECT CASE Message
    CASE _Data
      DDESelect(Session)
      SELECT CASE UCase$(Topic)
        CASE "Z1.ACME.HIGH"
          Acme.high = StrToNum(Data)
        CASE "Z1.ACME.LOW"
          Acme.low = StrToNum(Data)
        CASE "Z1.ACME.LAST"
          Acme.last = StrToNum(Data)
          IF Acme.last > Acme.high THEN
            Acme.high = StrToNum(Data)
            DDEControl(_Poke, "Z1.ACME.HIGH", \
              Data)
          ELSEIF Acme.last < Acme.low THEN
            Acme.low = StrToNum(Data)
            DDEControl(_Poke, "Z1.ACME.LOW", \
              Data)
          END IF
        END SELECT
      CASE _Close
        INPUT "Compete shut down";
      END SELECT
  END PROC

'Launch Compete.  Wait 5 seconds to be sure it has
'started
SHELL "C:\COMPETE\COMPETE STOCKS.MDL", _Inactive
IDLE 5

ClientSession = DDEQUnique
IF NOT(DDENew(ClientSession, "Compete", \
  "STOCKS.MDL")) THEN
  INPUT "Could not initiate DDE with Compete";
  EXIT PROGRAM
ELSE
  INPUT "Connected to Compete";
END IF

DDESetProc(ClientProc)
DDEControl(_Request, "Z1.ACME.HIGH")
DDEControl(_Request, "Z1.ACME.LOW")
DDEControl(_Advise, "Z1.ACME.LAST")
```


Once CA-Compete! is launched successfully, and the client procedure is set with `DDESetProc`, the client requests the high and low prices from CA-Compete!, found in cells `Z1.ACME.HIGH` and `Z1.ACME.LOW`, respectively. The client then asks to be advised of any changes to `Z1.ACME.LAST` (the last price).

The client procedure expects `_Data` messages for those three cells. When it receives the high and low values, it updates its own versions, *Acme.high* and *Acme.low*. When the client receives an update for the last price, it compares it to the high and low values, and pokes a new value back into the spreadsheet if a new high or low value has been reached.

Using CA-Realizer as a DDE Server

A CA-Realizer application can act as a DDE server for other applications. These applications can request and send data about any topic that the CA-Realizer server application wants to handle. A CA-Realizer server application can even handle DDE requests from another instance of CA-Realizer running as a client.

When an application attempts to initiate a DDE session, all other running applications, including CA-Realizer, receive an `_Initiate` message. In CA-Realizer, this message is automatically sent to the DDE procedure attached to the predefined server session. The identification number of the server session is always `-1`.

If your application wants to accept DDE messages and become a server, you should attach a Server Initiator procedure to the server session:

```
DDESelect(-1)
DDESetProc(ServerInitiator)
```

The Server Initiator procedure takes four parameters:

```
Proc ServerInitiator(Session, Message, App, Topic)
```

App is the application being sought, **not** the application that is requesting a conversation. Therefore, the initiator procedure should only respond if *App* is the name of the current application (unless it intentionally wants to answer DDE requests that were meant for other applications). *Message* is always `_Initiate` and *Topic* is the topic the requesting application wishes to discuss.

If the Server Initiator decides to accept the request for DDE, it must open a new session to handle the incoming requests. That is, the `_Initiate` message comes in as session -1, but any further messages go to the new server session that must be established. The server must use the *Session* parameter as an identifier for the new session and initiate it with `DDENew` as illustrated in the following example:

```
PROC ServerInitiator(Session, Message, App, Topic)  
  ' Message is always _Initiate  
  ' App is the application being sought  
  ' Topic is the topic they want to use  
  ' Only answer if the message is for us  
  IF App = "CA-Realizer" THEN  
    IF DDENew(Session, App, Topic) THEN  
      DDESetProc(ServerProc)  
    END IF  
  END IF  
END PROC
```

Once the new session is initiated to act as a server, a procedure designed to handle the incoming requests of the client must be attached to it. The general structure of this procedure is as follows:

```
PROC ServerProc(Session, Message, Topic, Data)
  IF Message <> _Close THEN
    DDESelect(Session)
  END IF
  SELECT CASE Message
    CASE _Request
      ' The client is requesting data on the
      ' topic. The data should be sent to the
      ' client with DDEControl(_Data).
    CASE _Advise
      ' The client is requesting that it be
      ' kept advised of any changes to the
      ' topic. The server should set a flag
      ' and send the client a _Data message
      ' any time the topic is updated.
    CASE _Unadvise
      ' The client is canceling a previous
      ' Advise. The server should note that
      ' _Data messages no longer need to be
      ' sent for this topic.
    CASE _Poke
      ' The client is sending data to the
      ' server. The server can use the data
      ' to update its values.
    CASE _Execute
      ' The client is sending a list of
      ' commands, in Topic, that it wants
      ' the server to execute.
    CASE _Close
      ' The client is shutting down and notifying
      ' the server. The server can clean any data
      ' relating to the session, such as an Advise
      ' flag. The server should not try to select
      ' or communicate with this session anymore.
  END SELECT
END PROC
```

The server procedure should respond to any messages that it wants to handle. For example, it could respond to _Data and _Advise requests and ignore _Poke and _Execute messages. It is important to select the session with the DDESelect command so that any DDE commands that follow are sent to the correct client. The exception to this is if the client has sent a message notifying the server that the client has closed the session—selecting a closed session creates an error.

Responding to Messages from a Client

After you receive a DDE message from a client, you need to send a response. You can use the following commands to respond to messages from a client:

```
DDEControl(_Close [, Topic [, Data]] [; Format])
```

```
DDEControl(_Data [, Topic [, Data]] [; Format])
```

DDEControl(_Data)

The DDE server procedure responds to the various incoming client messages by sending data back to the client with the DDEControl(_Data) command.

Topic is the topic of the data and *Data* is the actual information. If you specify *Format*, it indicates the format of the data being sent.

Data should be sent in the format requested by the client; otherwise, it may not be received. CA-Realizer, for example, ignores all non-text messages unless the session has specifically allowed them (as specified by the _AllFormats modifier in the DDENew command when the session was initiated).

Data to be sent in a non-text format should be packed in a string. The Asc, Chr\$, and Mid\$ functions can be used for format coding and decoding.

DDEControl(_Close)

There is only one other message the server can send. If the server is shutting down, it should notify all its clients by using the DDEControl(_Close) command for each client session. This closes all of the server's sessions and sends each client a message informing it that the connection is broken.

Obtaining Information about a DDE Session

When you need to obtain specific information about a particular DDE session, use the DDEQ function.

DDEQ(_Exists)

To determine whether a specific session exists, use the DDEQ(_Exists) function as follows:

```
Exists = DDEQ(_Exists [; SessionNum])
```

SessionNum is the number of the session whose existence you want to verify. If *SessionNum* is not specified, the currently selected session is used. If the session exists, its number is returned. If there is no current session, or the specified session does not exist, 0 is returned.

DDEQ(_Style)

The DDEQ(_Style) command indicates whether the session is a client or a server:

```
Style = DDEQ(_Style [; SessionNum])
```

SessionNum is the number of the session whose style you want to determine. If *SessionNum* is not specified, the currently selected session is used. If the session is a client, 1 is returned; if the session is a server, 2 is returned. If there is no current session, or the specified session does not exist, 0 is returned.

Using CA-RET with CA-REALIZER

CA-RET is a stand-alone, WYSIWYG visual report writer that can be used to create attractive and powerful reports based on data queries. When CA-RET is used as a DDE, CA-Realizer can be used to control the actions of CA-RET.

Note: A complete description of CA-RET's DDE capabilities and commands can be found in your CA-RET documentation.

Although the standard CA-Realizer DDE commands can be used to connect to CA-RET, control it, and query its data, CA-Realizer also includes a special command—CARET—which provides an easier method for direct DDE communications with CA-RET. The CARET command automatically launches CA-RET and maintains a DDE conversation; it is discussed in the next section.

Initiating a Conversation with CA-RET

To begin your interaction with CA-RET, you must first initiate a conversation with the CARET(_Start) command as follows:

```
CARET(_Start [, ReportName])
```

This command automatically loads the report specified by *ReportName*. If *ReportName* is omitted, the topic given to CA-RET is SYSTEM.

Executing a CA-RET Command

Once you have initiated a conversation, you can execute a CA-RET command with the CARET(_Execute) command as follows:

```
CARET(_Execute, Command)
```

This command tells CA-RET to execute *Command*. The valid DDE commands that can be sent to CA-RET can be found in your CA-RET documentation.

Requesting Information from CA-RET

You can request information from CA-RET using the `CARET(_Request)` function as follows:

```
Data = CARET(_Request, Item [, TimeOut])
```

This function requests specific information from CA-RET, which is returned as *Data*. The valid values for *Item* can be found in your CA-RET documentation. If *TimeOut* is specified, it indicates the number of seconds that CA-Realizer should wait for the reply.

Terminating the CA-RET Conversation

To terminate a CA-RET conversation, use the `CARET(_Stop)` command. `CARET(_Stop)` terminates the CA-RET DDE conversation, but does not close CA-RET.

To close CA-RET, use a `CARET(_Execute, "FILE.EXIT")` command before `CARET(_Stop)`.

A Sample CA-RET Conversation

The following example initiates a conversation with CA-RET, using the `BAR.RET` report. It then retrieves page 1, prints the entire report, closes CA-RET, and terminates the conversation.

```
'Initiate the conversation
CARET(_Start, "BAR.RET")

'Retrieve page 1 as text. Wait up to 10
'seconds
s = CARET(_Request, "PAGE1:1/TEXT"; 10)

'Have CA-RET print the entire report
CARET(_Execute, "FILE.PRINT(0,0)")

'Shut down CA-RET
CARET(_Execute, "FILE.EXIT()")
CARET(_Stop)
```


Chapter 26

External Procedures and Functions

In This Chapter	26-1
Dynamic Link Libraries	26-2
Declaring External Procedures	26-3
Passing Numeric Parameters	26-4
Passing Character Parameters	26-6
Passing Structure Parameters	26-7
Passing Pointer Parameters	26-9
Declaring External Functions	26-10
Accessing System DLL Functions	26-12
ODBC Interface Library	26-13

Chapter 26

External Procedures and Functions

In This Chapter

CA-Realizer can access procedures and functions written in any language (such as C, Pascal, COBOL, and Assembly language) that can create a *Dynamic Link Library* (or DLL).

You can use these external functions for special processing where extra speed is required, to link to third party *engine* libraries while still using CA-Realizer as a front end, or to port existing programs and libraries to Windows and OS/2 with minimal effort.

In this chapter, you will learn how to declare external procedures and functions, access Windows and OS/2 system DLL functions, and access the Open Database Connectivity (ODBC) interface common PI functions.

Note that all references to non-CA-Realizer functions are presented in *bold italic*.

Dynamic Link Libraries

Dynamic Link Libraries are libraries of functions to which Windows and OS/2 applications can link as they run—not as they compile. CA-Realizer programs can access the functions in any DLL, including the standard Windows and OS/2 DLLs.

This is a very powerful feature, because it gives CA-Realizer programs access to an almost unlimited library of additional routines. Furthermore, it lets CA-Realizer access any Windows or OS/2 function, providing an interactive programming environment for prototyping Windows and OS/2 programs written in lower-level languages, such as C.

For more information about DLLs, refer to your Windows and OS/2 documentation.

Important! Use caution when using external routines—misuse can crash or hang the system, and possibly delete or damage files.

Note: The code for a sample DLL written in C can be found in the REALIZER\SAMPLES\DLL directory.

Declaring External Procedures

You must declare external procedures in DLLs with an EXTERNAL PROC statement as follows:

```
EXTERNAL LibName PROC ProcName [(Type1 [Name1]  
    [... , TypeN [NameN]])] [ALIAS AliasName]
```

LibName is the name of the DLL. If the DLL is not in any of the paths listed in the CA-Realizer _MacroDir system variable, the PATH environment variable (for Windows), or the LIBPATH environment variable (for OS/2), you should specify the full path for the library file.

ProcName is the name that CA-Realizer uses to refer to the procedure—if the ALIAS option is not used, it is also the name of the procedure within the library.

If you specify *AliasName*, it represents the actual name of the procedure in the library, allowing the procedure to be renamed. *AliasName* can also be the ordinal number of the procedure within the library, as declared in the definition file of the library.

Note: It is faster to declare the procedure using its ordinal number.

The parameter list in the EXTERNAL statement must specify the type (*Type1...TypeN*) of each parameter, although you can omit the name (*Name1...NameN*) of the parameter. These types are detailed in the following sections of this chapter. Modifiers and optional parameters are not allowed.

There is no parameter count and type information within the DLL, so CA-Realizer cannot check these. You can, however, use an external function as a procedure if the return value can be ignored.

Passing Numeric Parameters

Numeric values of the following data types can be passed to external routines (note that parameters are cast automatically to fit the declared type when the procedure or function is called):

Type	Description
SINGLE	4-byte IEEE floating point value.
DOUBLE or REAL	8-byte IEEE floating point value.
LONG	4-byte signed integer.
DWORD	4-byte unsigned integer.
INTEGER	2-byte signed integer.
WORD	2-byte unsigned integer.
CHAR	1-byte signed value.
BYTE	1-byte unsigned value.
POINTER	If the value is a variable, its address is passed as a pointer. If the value is a constant or an expression, a pointer to a temporary version of the value is used. If the value is an array, a pointer to the array is passed.

POINTER parameter declarations can also include modifiers that force CA-Realizer to perform extra type checking. For example, POINTER (DOUBLE) would require a pointer to a real number, POINTER (STRING) would require a pointer to a string, and POINTER (INTEGER ARRAY) would require a pointer to an array of integers.

The ranges are checked as follows:

Type	Range
SINGLE	Unchecked
REAL or DOUBLE	Unchecked
LONG	-2,147,483,648 to 2,147,483,647
DWORD	Unchecked
INTEGER	-32768 to 32767
WORD	Unchecked
CHAR	-128 to 127
BYTE	Unchecked
POINTER	Unchecked

These types correspond to the following types in C:

CA-Realizer	C
SINGLE	float
REAL or DOUBLE	double
LONG	signed long
DWORD	unsigned long
INTEGER	signed int
WORD	unsigned int
CHAR	signed char
BYTE	unsigned char
POINTER	far *

Some examples of external declarations are shown below:

```
/* In DLL (Windows): */
void far Pascal Foo(int count, char ch,
    double far *retval);

'In CA-Realizer:
EXTERNAL "MyDLL" PROC Foo(INTEGER, CHAR, POINTER)

/* In DLL (OS/2) */
void AProcWithALongName(unsigned long x, double y)

'In CA-Realizer:
EXTERNAL "MyDLL" PROC Bar(DWORD, REAL) ALIAS \
    "AProcWithALongName"

/* In DLL (Windows): */
void far Pascal DivideArray(double far *array,
    int x)

'In CA-Realizer:
EXTERNAL "MyDLL" PROC DivideArray(POINTER, INTEGER)
```

Passing Character Parameters

Strings can be passed to external routines one character at a time, or as a far pointer to the string with the parameter types shown below:

Type	Description
CHAR	The first character of the string is passed as a 1-byte integer. This corresponds to the char type in C.
POINTER	A pointer to the string is passed as a pointer. This corresponds to a char far * in Windows and a char * in OS/2.

For example:

```
/* In DLL (Windows): */
WORD AnsiUpperBuff(char far *string,
    unsigned int count);

'In CA-Realizer:
EXTERNAL "user" PROC AnsiUpperBuff(POINTER, WORD)
```


Passing Structure Parameters

Entire structures can be passed as value parameters by storing the data in a string and passing the correct number of bytes. The structure can also be passed by reference by passing a pointer to a string containing the data. The `Chr$` and `String$` functions can be used to build up the data structure.

CA-Realizer records can also be created to take on the appearance of the data type in the DLL, and then be passed as a pointer.

Type	Description
STRUCTURE <i>NBytes</i>	<i>NBytes</i> of a string are copied to the stack without conversion. The external function must be expecting the correct number of bytes. This corresponds to any struct or union type in C.
POINTER	A pointer to the data structure is passed as a far pointer. This corresponds to any pointer type in C.

For example, the Windows `RECT` structure is defined as:

```
typedef struct tagRECT {
    int left;
    int top;
    int right;
    int bottom;
} RECT;
```

The Windows function, *SetRect*, takes a pointer to a `RECT` structure and four integer parameters to store in the structure:

```
void SetRect(LPRECT lpRect, int X1, int Y1,
            int X2, int Y2);
```

You can find *SetRect* in the Windows USER library. The CA-Realizer declaration for this procedure is as follows:

```
EXTERNAL "user" PROC SetRect(POINTER, INTEGER, \
    INTEGER, INTEGER, INTEGER)
```

You can create the space for an empty RECT structure with the `String$` function. For example, the following example defines an 8-byte RECT structure filled with 0's:

```
MyRect = String$(8, 0)
```

You can then call *SetRect* to fill in the values as follows:

```
SetRect(MyRect, 50, 50, 200, 100)
```

MyRect can then be passed to any function expecting a RECT structure.

A better solution is to create a CA-Realizer record type that is identical to the structure declared in C. For example:

```
TYPE Rect
    left AS INTEGER
    top AS INTEGER
    right AS INTEGER
    bottom AS INTEGER
END TYPE
```

A variable of type *Rect* can then be created with the DIM statement:

```
DIM MyRect AS Rect
```

MyRect can be passed to the *SetRect* function in the same way as before, but this method makes it easier to extract the values. For example, if the previous call to *SetRect* were used, the values of *MyRect.left*, *MyRect.top*, *MyRect.right*, and *MyRect.bottom* would be 50, 50, 200, and 100, respectively.

Passing Pointer Parameters

You can use the `POINTER` parameter type to pass the address of any variable or value. If you need an actual pointer value, use the `CA-Realizer Pointer` function as follows:

```
ns = Pointer(Pointer)
```

The most common use is for passing `NULL` pointers. For example, consider the following OS/2 procedure, which expects a pointer and a `CA-Realizer` external declaration:

```
void AddTen(double far *x)
{
    if (x != NULL)
        *x = *x + 10;
}
```

```
EXTERNAL "MyDLL" PROC AddTen(POINTER)
```

If you use a numeric variable with the procedure, a pointer to that variable is passed and the value of the variable is increased by 10, as shown below:

```
Z = 0
AddTen(Z)      'A pointer to the value of Z
```

If you use an expression or constant, a pointer to a temporary version of the value is passed. The value is first increased by 10, and then immediately thrown out by `CA-Realizer`:

```
AddTen(0)      'A pointer to a temporary value 0
```

If you use the `Pointer` function, the value is passed as the actual pointer value:

```
AddTen(Pointer(0))      'A NULL pointer
```

You can also use the `Pointer` function to pass pointer values that were returned by other external procedures or functions, as explained later in this chapter.

Declaring External Functions

External functions are identical to external procedures except that external functions return a value. In fact, the only difference between a CA-Realizer external procedure and a CA-Realizer external function is that CA-Realizer ignores the return value for an external procedure.

To declare an external function, use the `EXTERNAL FUNC` command as follows:

```
EXTERNAL LibName FUNC FuncName [(Type1 [Name1]  
    [..., TypeN [NameN]])] AS ReturnType  
    [ALIAS AliasName]
```

LibName is the name of the DLL. If the DLL is not in any of the paths listed in the CA-Realizer `_MacroDir` system variable, the `PATH` environment variable (for Windows), or the `LIBPATH` environment variable (for OS/2), you should specify the full path for the library file.

FuncName is the name that CA-Realizer uses to refer to the function—if the `ALIAS` option is not used, it is also the name of the function within the library.

If you specify *AliasName*, it represents the actual name of the function in the library, allowing the function to be renamed. *AliasName* can also be the ordinal number of the function within the library, as declared in the definition file of the library.

Note: It is faster to declare the function using its ordinal number.

ReturnType is the type of the return value. Functions can return numeric values as any of the following types, all of which are converted to CA-Realizer numeric values:

Type	Description
SINGLE	4-byte IEEE floating point value.
DOUBLE or REAL	8-byte IEEE floating point value.
LONG	4-byte signed integer.
DWORD	4-byte unsigned integer.
INTEGER	2-byte signed integer.
WORD	2-byte unsigned integer.
CHAR	1-byte signed value.
BYTE	1-byte unsigned value.
STRUCTURE <i>NBytes</i>	<i>NBytes</i> are copied to the stack without conversion. The external function must be expecting the correct number of bytes.

Structures can be returned directly off the stack. *NBytes* are copied and stored as a string. You can then use the Mid\$ and Asc functions to access the data:

```
STRUCTURE NBytes
```

Functions that return pointers should have a return type of DWORD. Use the CA-Realizer Pointer function to pass this pointer value back to a function requiring a pointer.

Accessing System DLL Functions

You can use CA-Realizer to access any function in a Windows or OS/2 system DLL. These functions provide convenient functionality, direct access to the system interface tools, and complex system management features.

Some of these functions use *handles*. Many require a handle to a window, memory, or a graphic object such as a bitmap. In these cases, you can treat the handles as WORDs (unsigned 2-byte integers). An external function can return a handle that can then be passed back to another function.

The ability to manipulate handles allows complete access to Windows and OS/2. You can even use CA-Realizer as a prototyping tool for native system development with instant turnaround.

Custom controls provide an even more sophisticated method of integrating Windows code written in C—or other languages—with CA-Realizer programs. Custom controls allow a window to be created and controlled by CA-Realizer as an object in a form, but all the processing, painting, and message handling in the window is left to a window procedure written in C. For more information about custom controls, see the “Custom Controls” chapter in this guide.

Refer to your Windows or OS/2 documentation for descriptions of the system library functions.

ODBC Interface Library

The *Open Database Connectivity* (ODBC) interface is a protocol for accessing a wide variety of different database drivers using a common *Application Program Interface* (API). This API consists of a series of functions divided into three categories: *Core*, *Level 1*, and *Level 2*. These functions can be accessed directly by CA-Realizer as external functions—CA-Realizer includes a library that predeclares these functions.

To access an ODBC driver, you must include two statements in your CA-Realizer application. The first loads the ODBC interface:

```
RUN "ODBC"
```

The interface, ODBC.RLZ, is located in the REALIZER\LIB directory, which is automatically searched as part of the `_MacroDir` path list. When ODBC.RLZ is run, it declares the *InitODBC* function:

```
InitODBC(DriverName, Level)
```

A call to *InitODBC* should be made before using any of the ODBC functions. There are two required parameters: *DriverName* and *Level*. *DriverName* is the name of the ODBC DLL, and should be specified with quotes. If the ODBC DLL is not in the path, *DriverName* should also include the full directory.

Level is the ODBC level that is implemented by the DLL—either 0 (core only), 1 (core and level 1), or 2 (core, level 1, and level 2).

Once the interface has been initialized, any of the functions for the ODBC level declared can be used. Refer to the ODBC API documentation for more information.

Chapter 27

Custom Controls

In This Chapter	27-1
What Is a Custom Control?	27-2
Declaring Custom Controls	27-3
Placing a Custom Control in a Form	27-4
Adding Custom Controls to FormDev	27-5
Creating Custom Control Libraries	27-5
Defining Constants	27-5
Registering Custom Controls	27-6
Custom Control Options	27-8
Option Value Fields	27-8
Options Dialog Function	27-8
Options Generation Function	27-9
Options Importation Function	27-9
Creating a Custom Control DLL	27-10
Windows and OS/2 Messages	27-10
CA-Realizer Notification Messages	27-12
CA-Realizer Query Messages	27-12
CA-Realizer Numeric Value Messages	27-14
CA-Realizer String Value Messages	27-15
Order of Messages	27-17
Custom Messages	27-18
Sending Messages to CA-Realizer	27-19
Posting a Message to the Form Procedure	27-19
Sending a Message to the Form Procedure Immediately	27-20
Style Flags	27-20
Form Colors in Windows	27-21

Chapter 27

Custom Controls

In This Chapter

CA-Realizer's custom controls provide the ability to incorporate a set of specialized GUI controls (both form objects and windows) into your CA-Realizer programs. By creating a library of custom controls, you can import these specialized controls into the FormDev environment so that they can be used in your FormDev projects like any of the standard CA-Realizer form objects and tools.

At the most basic level, a custom control is a user-defined form object, such as a gauge, meter, or a new type of button. On a larger scale, custom controls can be an entire window that is controlled by an external program. The painting, message processing, and storage management of the window are all handled by the custom control, which allows entire tools to be created and seamlessly added to CA-Realizer as a library.

This chapter describes how custom controls are created and used. The creation of a custom control requires detailed understanding of both C and Windows or OS/2 programming (refer to the appropriate compiler documentation for more information). The use of a custom control does not require any special knowledge, although the specifics of the control should be documented separately by its creator.

Note that all references to both the FormDev functions and non-CA-Realizer functions are presented in *bold italic*.

What Is a Custom Control?

In CA-Realizer, custom controls are the same as any other form object. They are created with the `FormSetObject` command and manipulated with the `FormModifyObject` command, and they can be interacted with either passively or actively. A custom control can contain text and numeric values, and it can notify the form to invoke a form procedure or to satisfy a `FormWait`.

From the Windows or OS/2 programmer's perspective, a custom control is an instance of a window class that the programmer defines. The `FormSetObject` command creates the window, but the window procedure handles the rest of the processing. CA-Realizer passes along all the relevant messages, as well as CA-Realizer-specific messages.

When adding custom controls, CA-Realizer handles the interface syntax for you—for example, use CA-Realizer's position notation to specify the size and position, and CA-Realizer will create the window accordingly. If a font is specified in the `FormSetObject` command, the control gets a handle to that font. The control can even respond to customized messages that are sent with the `FormModifyObject` command.

As stated previously, a control can be as simple as a slider or as complex as an entire window. You can manipulate such controls, and then poll their values (for example, to check the status of a check box or option button).

When a custom control encompasses an entire window, the control can maintain large data structures, handle complex paint processing, and communicate with a full set of messages. Such controls are actually full programs that revolve around a main window, just like the CA-Realizer tools.

Declaring Custom Controls

Before using a custom control in a form, you must declare the Dynamic Link Library (DLL) in which it is contained. To do this, use the `EXTERNAL CUSTOMCONTROLS` command as follows:

```
EXTERNAL DLLName CUSTOMCONTROLS
```

This command informs CA-Realizer that the DLL specified by *DLLName* contains the code for one or more custom controls. (You do not have to indicate the names of the controls in the library.)

CA-Realizer searches for the specified DLL in the paths specified in the `_MacroDir` system variable. If it is not found there, the `PATH` environment variable (under Windows) or the `LIBPATH` environment variable (under OS/2) is searched. (If the DLL is not in any of these paths, specify the complete path as part of *DLLName*.)

Note: The creation of a custom control DLL is discussed in greater detail later in this chapter.

Placing a Custom Control in a Form

FormSetObject

Like standard form objects, you can use the `FormSetObject` command to add custom controls to a form. The only difference is that the type of the control is specified with a text string (instead of a predefined constant like `_Button` or `_ListBox`).

The specified string must contain the name of the custom control, which is the name of the control's window class:

```
FormSetObject(ID, CustCtrlName, Text,  
             [Font,] Left, Top [, Width, Height]  
             [, Mod1 [..., ModN]])
```

The usage of *Font*, *Text*, and the modifiers *Mod1...ModN* depends on how the control was designed and how it functions.

For example, consider a custom control called `CC_Slider`, located in a DLL named `CCLIB`, that creates a horizontal slider object. It would be placed in a form as follows:

```
EXTERNAL "CCLib" CUSTOMCONTROLS  
FormNew  
FormSetObject(10, "CC_Slider", "", _Center, \  
              20 pct, 80 pct, _Default)
```

CA-Realizer's position notation specifies the size and position of the control. When not specified explicitly, CA-Realizer asks the custom control for its default size and then calculates the position and size of the window accordingly.

Adding Custom Controls to FormDev

In addition to adding custom controls to a form using the `FormSetObject` command, you can customize the FormDev object palette by adding icons to it that represent your custom controls. When added to the object palette, a custom control can be placed in your forms like any other form objects.

By choosing the `ADD CUSTOM CONTROL` command from the FormDev UTILITIES menu, you can specify the name of the custom control library (.FCC) which contains the controls to be added to the object palette. See *Creating Custom Control Libraries* for instructions on how to create .FCC files.

Tip: CA-Realizer provides a sample custom control library named `OBJ3D.FCC` in the `REALIZER\LIB` directory.

Creating Custom Control Libraries

A custom control library file contains CA-Realizer program source code that registers the control with FormDev, and provides CA-Realizer with procedures that allow the you to specify options for the object.

Defining Constants

The first thing that the library file should do is `RUN` the associated library module (for example, `RUN "OBJ3D"`). This serves to define the constants associated with the custom control, so that the constants can be used during registration of the custom control with FormDev.

Registering Custom Controls

The custom controls are then registered with FormDev by calling the FormDev function, *FormDevInstallControl*, which takes a family as its only parameter. This family is expected to have the following specific members in it, which provide all of the necessary information to FormDev:

Control.Bitmap

A string containing the path and file name of a bitmap. This bitmap will be used to represent the control with which it is associated in the FormDev object palette.

This bitmap should be 76 x 20 pixels (width and height)—the same size as all other bitmaps in the object palette—and should contain two side-by-side pictures. The left half is the off (or unclicked) state; the right half is the on (or clicked) state.

Tip: It might be easiest to start with one of the CA-Realizer bitmaps (for example, GRP3D.BMP) when creating your own.

Control.DLL

A string containing the name of the .DLL file, without the .DLL extension (for example, "OBJ3D").

Control.Name

A string containing the type name used by the FormSetObject command. For example, if the type name is _Group3D, this field would be "_Group3D" (quotes are required). This name is used to generate code and test the form.

Control.TextName

A string which represents the name of this object type (for example, "3DGroup"). This string is used in the type name list box.

Control.FDOptionsFunc

A string containing the name of the function to call when you ask to change the modifiers or initializers of this object type. If there are no modifiers for this object type, this field should be set to an empty string. For more information about this function, see Custom Control Options.

Control.FDGenerateFunc

A string containing the name of the function to call to generate the modifier string for this object type. If there are no modifiers for this object type, this field should be set to an empty string. For more information about this function, see Custom Control Options.

Control.FDReadFunc

A string containing the name of the function to call that will interpret a string representation of the modifiers if this object type is imported. If there are no modifiers for this object type, this field should be set to an empty string. For more information about this function, see Custom Control Options.

**Control.Val1,
Control.Val2,
Control.Str**

Two numeric fields and one string field that you can use to store modifier values. Modifiers are discussed later in this chapter.

When all of these members have been initialized, you should call *FormDevInstallControl(Control)* to register the control. Multiple controls can be registered in the same .FCC file.

Custom Control Options

Many of the objects in FormDev have options that allow you to customize the look or operation of the object. For example, when working with a check box, you can specify its initial state and whether it notifies the form. For list boxes, you can specify their initial contents.

A custom control can also have options. What the options are, and how they are used, is left entirely up to you. FormDev provides hooks to its storage of these options (also called *modifiers*, since they become the modifier values in the resulting FormSetObject command), but requires that you create three functions and include them in the .FCC file.

Option Value Fields

The values for the options are stored in the numeric and string values *Val1*, *Val2*, and *Str*. For Boolean or mutually exclusive values, several values can be packed together using bit fields and the bitwise functions (BitAnd, BitOr, BitSet, and BitTest). And, since the string value can contain numeric information converted to binary with the MK\$ and CV function, the control can store literally 64K worth of information.

These fields should initially be set to the defaults for the object type.

Options Dialog Function

The function specified by *Control.FDOptionsFunc* is called when you ask to change the options for an object type, and displays a dialog box that allows you to set the options for this object type. This function is called with a single parameter, which is a family that contains the three members—*Val1*, *Val2*, and *Str*. These members contain the current value of the object's options. When you finish changing the options, this function returns a family containing these same fields with the updated values.

Options Generation Function

The function specified by *Control.FDGenerateFunc* is called whenever FormDev needs to generate the modifiers for this object type. This function is called with a single parameter, which is a family that contains three members—*Val1*, *Val2*, and *Str*. These fields are set to the object's current options values.

This function returns a string representation of the modifiers that would follow the semicolon and come before the closing parenthesis of a *FormSetObject* or a *FormModifyObject* command. For example, if a *FormSetObject* command appears as follows,

```
FormSetObject(mygroup, _Group3d, "MyGroup", \\  
             left, top; _3dLeft)
```

the function returns the string "_3DLeft". If no modifiers are needed, this function returns an empty string.

Options Importation Function

The function specified by *Control.FDReadFunc* is called whenever FormDev is importing an object of this type. This function is called with a single parameter—an array of strings. This array of strings is a tokenized representation of all of the modifiers following the semicolon up to, but not including, the closing parenthesis of a *FormSetObject* command.

This function parses these values, interprets them, and returns a family that contains the three options members: *Val1*, *Val2*, and *Str*.

For more information about using custom controls with FormDev, refer to the *CA-Realizer User Guide*. To view a sample custom control library, take a look at the OBJ3D.FCC file in the REALIZER\LIB directory.

Creating a Custom Control DLL

A custom control is a window class, declared and registered in the DLL's initialization function (*LibMain* in Windows and *_DLL_InitTerm* in OS/2). The class will have a window procedure that handles all of the control's painting and message processing.

Protocols

There are several protocols that CA-Realizer expects to be maintained. Note that there is not a great deal of difference between the window procedure for a custom control and that for any other window. A custom control can respond to any of the Windows or OS/2 messages it chooses, as well as a few special CA-Realizer messages.

The Header File

To compile, custom controls require the header file *CUSTCTRL.H*, which is located in the *REALIZER\CUSTCTRL* subdirectory (along with other sample custom control libraries.)

Windows and OS/2 Messages

As stated, a custom control's window procedure can respond to any Windows or OS/2 message. However, it should provide default values for the most common messages, some of which are sent by CA-Realizer when the control is created or modified.

The return value from a window procedure is a long integer. Refer to your Windows and OS/2 documentation for a complete description of the following messages.

WM_CREATE

This message is sent when the custom control window is created.

WM_DESTROY

This message is sent when the control is destroyed.

WM_SETFONT In Windows, this message is sent immediately after the custom control window is created if a font is specified in the `FormSetObject` command. The custom control receiving this message **must not** destroy the font.



In OS/2, the font is automatically set before you paint, so this message is not necessary.

WM_ENABLE This message, which is sent to enable or disable mouse and keyboard input, occurs when a form is brought to the front or sent to the back, or when a `FormModifyObject` command is used to gray or ungray a custom control.

WM_GETTEXT
WM_GETTEXTLENGTH These messages are sent in response to a `FormQStr` function for a single string.



Under OS/2, these messages are sent as `WM_QUERYWINDOWPARAMS` messages with *ulStatus* values of `WPM_TEXT` and `WPM_CCHTEXT`, respectively.

WM_SETTEXT This message is sent to change a custom control's name or title, as specified with the `FormSetObject` or `FormModifyObject` command.



Under OS/2, this message is sent as `WM_SETWINDOWPARAMS` with *ulStatus* set to `WPM_TEXT`.

Note: In both Windows and OS/2, it is not absolutely necessary to process the `WM_GETTEXT`, `WM_GETTEXTLENGTH`, and `WM_SETTEXT` messages. By default, CA-Realizer will maintain the text value by itself if you allow the messages to be handled by the default window procedure. The control can handle them, however, if it needs to override this default and process the text string. To query the current value of the default text string, use `GetWindowText(hwnd)` in Windows and `WinQueryWindowText(hwndParent)` in OS/2.

CA-Realizer Notification Messages

There are a number of custom messages that CA-Realizer sends to a control both in response to various actions and to retrieve information from the control.

The two message parameters are referred to in this section as *Param1* and *Param2*. In Windows, they are generally referred to as *wParam* and *lParam*, while in OS/2, the names *mpParam1* and *mpParam2* are used.

CCM_FORMSIZED This message is sent after the form containing the custom control is resized (if that form is not minimized). The custom control can then change its position.

For this message, *Param1* and *Param2* are not used.

CCM_RESETCONTENT This message is sent when the program uses the `FormModifyObject(_Clear)` command.

For this message, *Param1* and *Param2* are not used.

CA-Realizer Query Messages

CCM_QDEFAULTSIZE The `CCM_QDEFAULTSIZE` message is sent after the custom control has been created and the `FormSetObject` modifiers have been sent, but before the object has been shown or enabled. It is a request by CA-Realizer for the control to specify its default size. A `WM_SETFONT` message, if any, is sent before this message.

For this message, *Param1* and *Param2* are not used.

If a custom control wants to specify a default size, it should pack the width and height (in pixels) into a long integer with the width in the low-order word and the height in the high-order word. (In Windows, this can be done using `MAKELONG(width, height)`).

Otherwise, you can use one of the following values for CCM_QDEFAULTSIZE:

Value	Meaning
0L	Use the default size of a normal button.
-1L	Destroy the control and report an error.
-2L	Do not resize the control.
-3L	Use the full size of the form.

CCM_GETFLAGS

This message is sent immediately after CCM_QDEFAULTSIZE to get the style flags for the custom control.

For this message, *Param1* and *Param2* are not used.

The return value is 0, or a sum of the following flags that indicate the style of the custom control:

Flag	Meaning
CCF_FOCUS	The custom control will take the focus. If this flag is set, the WM_SETFOCUS and WM_KILLFOCUS messages are processed. The focus should also be visibly apparent and is required to receive keyboard messages, but it is not required to receive mouse messages.
CCF_INITGRAY	The custom control is initially disabled.
CCF_NOENABLES	The custom control should not be enabled and disabled with the form. If this flag is set, the control only gets WM_ENABLE messages when it is grayed or ungrayed.

CA-Realizer Numeric Value Messages

Custom controls can maintain numeric values that you set with `FormModifyObject` and query with `FormQNum` or `FormQObject`. For example, the value of a slider control would be the position of the slider in some specified units. A control, such as a check box or an option button, would hold a value of 0 or 1 (on or off). The value must be a long integer.

When you use `FormModifyObject` with numeric modifiers, CA-Realizer sends one or more `CCM_SETNUM` messages to the control. When you query the value with `FormQNum` or `FormQObject`, a `CCM_GETNUM` message is sent.

CCM_GETNUM

The `CCM_GETNUM` message is sent when you use `FormQNum` or `FormQObject` to obtain the numeric value of the custom control.

For this message, *Param1* and *Param2* are not used.

The return value is a long integer representing the current value of the custom control.

CCM_SETNUM

The `CCM_GETNUM` message is sent in response to a `FormSetObject` or `FormModifyObject` with real scalar modifiers. The values are converted to long integers.

The following table shows parameters for this message:

Parameter	Description
<i>Param1</i>	The number of consecutive real scalar modifiers.
<i>Param2</i>	A pointer to an array of long integers containing the values.

The return value is 1 if successful or -N, which indicates that the value in *Param2*[N] could not be processed.

CA-Realizer String Value Messages

Custom controls can maintain one or more string values. An edit control, for instance, maintains a single line of text, and a list box maintains a list of strings.

Using a Single String

When only one string is used, it is called the *name* or *title* of the control. It can be set by specifying *Text* with the `FormSetObject` and `FormModifyObject` commands, which send a `WM_SETTEXT` to the control. You can query the text by using the `FormQStr` command without *Start*, which sends `WM_GETTEXTLENGTH` and `WM_GETTEXT` messages.

Using a List of Strings

When a list of strings are maintained, you can set the strings by listing them as modifiers—either alpha scalars or alpha arrays—in `FormSetObject` and `FormModifyObject`. This produces a set of `CCM_SETSTR` messages that the control should process.

When you query the string list by using `FormQStr` and specifying *Start*, the control first receives a `CCM_GETSTRCOUNT` to find out how many strings there are, and then it receives a series of `CCM_GETSTRLEN` and `CCM_GETSTR` messages.

CCM_GETSTRCOUNT

The `CCM_GETSTRCOUNT` message is sent when you use the `FormQStr` command and specify *Start*. This assumes that the custom control is maintaining a list of strings numbered from 1 to N. The message will be followed by one or more `CCM_GETSTRLEN` and `CCM_GETSTR` message combinations.

When `FormQStr` is used without *Start*, the control receives `WM_GETTEXT` and `WM_GETTEXTLENGTH` instead.

For this message, *Param1* and *Param2* are not used.

The return value is the number of strings in the list.

CCM_GETSTRLEN

The CCM_GETSTRLEN message is sent prior to a CCM_GETSTR message to retrieve a string from the control as a response to a FormQStr command with *Start* specified.

Note: Do not confuse this with WM_GETTEXTLENGTH, which is associated with the title of the control.

The following table shows parameters for this message:

Parameter	Description
<i>Param1</i>	The index of the desired string, ranging from 1 to the value returned in response to the CCM_GETSTRCOUNT message.
<i>Param2</i>	Is not used.

The return value is the number of bytes needed to hold the desired string. This value does not include a null terminator.

CCM_GETSTR

The CCM_GETSTR message is sent immediately after a CCM_GETSTRLEN to obtain one of the string values of the custom control in response to a FormQStr command with *Start* specified.

Note: Do not confuse this message with a WM_GETTEXT message, which is associated with the title of the control.

The following table shows parameters for this message:

Parameter	Description
<i>Param1</i>	The index of the desired string, ranging from 1 to the value returned in response to the CCM_GETSTRCOUNT message.
<i>Param2</i>	Pointer to a buffer large enough to receive the string. The control copies <i>n</i> bytes into the buffer, where <i>n</i> is the value returned in response to the preceding CCM_GETSTRLEN message. Null termination of the string is optional.

CCM_SETSTR

The CCM_SETSTR message is sent when the program executes a FormSetObject or FormModifyObject with string modifiers. String scalars generate a single CCM_SETSTR message with *Param1* set to 0, and string arrays generate a set of CCM_SETSTR messages with *Param1* traversing the valid range of the array.

The following table shows parameters for this message:

Parameter	Description
<i>Param1</i>	0 for a string scalar, or the element number of the string in a vector.
<i>Param2</i>	A pointer to the null terminated string or array element.

The return value is 1 if successful; otherwise, 0 is returned.

Order of Messages

The initial messages are sent to the custom control in the following order:

- WM_CREATE
- WM_SETFONT

Messages for the FormSetObject modifier values are sent to the custom control in the following order:

- CCM_SETNUM
- CCM_SETSTR
- CCM_QDEFAULTSIZE
- CCM_GETFLAGS

Custom Messages

A custom control can receive other messages that are specific to the control. You send the message with the `FormModifyObject` command by using an *Attrib* value of `_CustomMessage`:

```
FormModifyObject(ID, _CustomMessage [...];  
                Msg, Param1, Param2)
```

The message is sent after any `WM_SETTEXT`, `WM_MOVE`, or `WM_SIZE` messages that are sent if other parameters are specified in the command.

Msg is the number of the message as defined by the control. Define this number as `_CCM+n`, where *n* is a small positive integer. The `CUSTCTRL.H` header file defines the constant `CCMBASE`, which matches CA-Realizer's built-in constant `_CCM`.

Param1 is passed to the window procedure as *Param1*. *Param2* is passed to the window procedure as *Param2*. The value and type of *Param2* depends on the type of value used for *Param2*, as shown in the following table:

Type <i>Param2</i>	Type Used for <i>param2</i>
Numeric scalar	The value is rounded and passed as a signed long integer
Numeric array	A pointer to the array is passed as a double *
String scalar	A pointer to the string is passed as a char *
String array	A pointer to the array is passed as a char **

In Windows, all pointers are far pointers.

The custom control returns a non-zero value to indicate success. A zero value causes CA-Realizer to report an error.

Sending Messages to CA-Realizer

There are two ways for the custom control to send a message to its form and invoke the form procedure. In one case, the message is queued with other messages, and waits for the program to become idle before being invoked. In the other case, the program's execution is suspended and the form procedure is called immediately.

The messages should be sent to the form, which is the parent of the control. Use the Windows *GetParent* function or the OS/2 *WinQueryWindow* function to find the form's window handle. Use the Windows *SendMessage* function or the OS/2 *WinSendMsg* function to send the message to the form.

Posting a Message to the Form Procedure

In most cases when the control wants to notify the form, it will want to do so in the same way that mouse clicks and other events are handled. If the control sends a CCM_CUSTOM message to the form, the form procedure is invoked when the program becomes idle. If a FormWait is pending, the ID of the control is returned.

The value of *params*[_Invoke] in the form procedure will be *_Custom*. The value of *params*[_CustomWP] will be equal to the message's *Param1*. The value of *params*[_CustomLP] will be equal to the message's *Param2*.

The value returned by the *SendMessage* or *WinSendMsg* call is 1 if the message was posted; otherwise, 0 is returned, indicating that the message queue was full.

Sending a Message to the Form Procedure Immediately

If a `CCM_CUSTOM_NOW` message is sent to the parent, CA-Realizer suspends the program's execution (if there is any active program, and then only after the current statement has been completed) and invokes the form procedure.

The value of `params[_Invoke]` in the form procedure will be `_CustomNow`. The value of `params[_CustomWP]` will be equal to the message's `Param1`. The value of `params[_CustomLP]` will be equal to the message's `Param2`.

The value returned by the `SendMessage` or `WinSendMsg` call is 1 if the message was posted; otherwise, 0 is returned, indicating that the message queue was full.

Style Flags

Some controls need to accept optional style flags (the CA-Realizer list box, for example, can use the `_Sorted` modifier). A control that accepts real values can differentiate a value representing style flags by specifying them as `_CCStyle + flag1 + flag2 + ...`, etc. The custom control can then compare the first real value to the `CCSTYLE` constant in `CUSTCTRL.H` to determine if the style flags were used.

This is only a convention and will **not** work if the custom control allows other 32-bit parameters.

Form Colors

Custom controls should behave like other form objects—for example, they should draw themselves using the text and field colors set by with the `FormSetColor` command. The following pieces of code can be used at the start of the `WM_PAINT` processing to get these colors from CA-Realizer.



In Windows, use the following:

```
HBRUSH hBgndBrush;
hBgndBrush = (HBRUSH)SendMessage(GetParent(hWnd),
    WM_CTLCOLOR, ps.hdc, MAKELONG(hWnd,
    CTLCOLOR_STATIC));
FillRect(ps.hdc, &ps.rcPaint, hBgndBrush);
```

When the control sends a `WM_CTLCOLOR` message to the form, the *hdc* for the control is passed as one of the parameters. CA-Realizer takes this handle and sets the text color (with *SetTextColor*), the background (field) color (with *SetBkColor*), and returns a handle to a brush that can be used to draw the background.



In OS/2, use the following:

```
CHARBUNDLE cbun;
ULONG AttrFound, BackGrColor, ForeGrColor;
// Query & set background (_Field) and Foreground
// (_Text)
// colors
WinQueryPresParam(hWnd, PP_BACKGROUND_COLOR, 0,
    &AttrFound,
    sizeof(BackGrColor), &BackGrColor,
    QPF_PURERGB_COLOR);
if (PP_BACKGROUND_COLOR == AttrFound) {
    cbun.lBackColor = (LONG)BackGrColor;
    GpiSetAttrs(ps->hdc, PRIM_CHAR, CBB_BACK_COLOR,
        0L, &cbun);}

WinQueryPresParam(hWnd, PP_FOREGROUND_COLOR, 0,
    &AttrFound,
    sizeof(ForeGrColor), &ForeGrColor,
    QPF_PURERGB_COLOR);
if (PP_FOREGROUND_COLOR == AttrFound) {
    cbun.lColor = (LONG)ForeGrColor;
    GpiSetAttrs(ps->hdc, PRIM_CHAR, CBB_COLOR, 0L,
        &cbun);}
```


Chapter 28

Encrypting CA-Realizer Programs

In This Chapter	28-1
Saving Encrypted Files	28-1
The Scope of Encrypted Symbols	28-2
Exporting Symbols	28-2
Local Symbols	28-4
Clearing Exported Symbols	28-5

Chapter 28

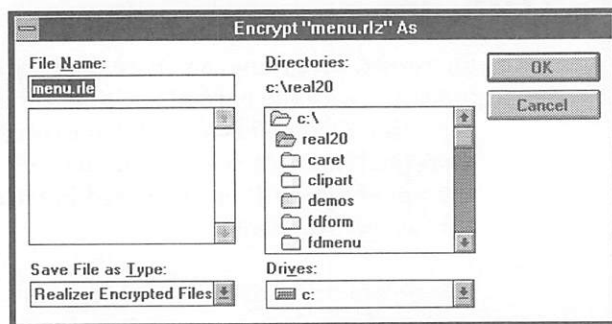
Encrypting CA-Realizer Programs

In This Chapter

You can encrypt your CA-Realizer programs to inhibit user access to source code or any procedures or variables created by the program. This feature protects developers who wish to sell or share their programs with others, but do not want the recipients to have unrestricted access to the program or its data.

Saving Encrypted Files

To save a CA-Realizer file in an encrypted form, press ALT+F1. The following dialog box is displayed, prompting you for the name of the file to encrypt:



Important! You can execute and RUN encrypted files, but you cannot open them. If an error occurs in an encrypted file, CA-Realizer displays the line number and error message, but not the source code line. Once a file has been encrypted, it cannot be read back into a window; therefore, it is advisable to keep an unencrypted version of the program in case you need to examine or modify it.

The Scope of Encrypted Symbols

As far as the user is concerned, variables, procedures, and functions in encrypted files do not exist. Symbols with the same name can be used in the normal (*outside*) environment without affecting those in the encrypted (*inside*) program. Likewise, the encrypted program will not have access to symbols in the normal environment. Encrypted symbols do, however, use the same amount of memory as normal symbols.

For example, if an encrypted file creates a variable named *MyVar*, the instant line

```
PRINT MyVar
```

will not show the value of the variable, but instead will report an error for an undefined symbol. Likewise, setting *MyVar* to a value outside of the encrypted program will not affect the value of *MyVar* in the encrypted file. Those variables, procedures, and functions created by encrypted files will not appear in the Variable Display list.

Exporting Symbols

Encrypted programs can share their variables, procedures, and functions with the normal CA-Realizer environment by exporting them. This allows you to access procedures in encrypted libraries, the user to modify variables in the encrypted file, and the encrypted file to see variables from the normal environment.

The general form of the EXPORT command is:

```
EXPORT Name1 [... , NameN]
```

Name1...NameN is a list of variables, procedures, and functions. The scope of an exported symbol depends on when and where it was first defined—*inside* or *outside*.

Inside refers to the encrypted file, and *outside* refers to any unencrypted file or instant command. CA-Realizer will ignore an EXPORT statement in a non-encrypted file.

If the symbol exists inside but not outside at the time of the EXPORT, the outside will see the current encrypted value:

ENCRYPT1.RLE:

```
Foo = 57.2
EXPORT Foo
PRINT Foo
```

USER1.RLZ:

```
RUN "ENCRYPT1.RLE"
PRINT Foo
```

The results will be as follows:

```
57.2000
57.2000
```

If the symbol exists outside but not inside at the time of the export, the inside will see the current unencrypted value:

ENCRYPT1.RLE:

```
PROC PrintFoo
    PRINT Foo
END PROC

EXPORT PrintFoo
EXPORT Foo
```

USER1.RLZ:

```
RUN "ENCRYPT2.RLE"

Foo = 11.9
PRINT Foo
PrintFoo
```

The results will be as follows:

```
11.9000
11.9000
```

If the symbol exists in both places at the time of the EXPORT, the outside version is destroyed, and the inside version is seen by both the inside and the outside:

ENCRYPT3.RLE:

```
Foo = "Blah"
PRINT "Inside before export:", Foo
EXPORT Foo
PRINT "Inside after export:", Foo
```

USER3.RLZ:

```
Foo = 123
PRINT "Outside before export:", Foo
RUN "ENCRYPT3.RLE"
PRINT "Outside after export:", Foo
```

The results will be as follows:

```
Outside before export: 123.0000
Inside before export:  Blah
Inside after export:   Blah
Outside after export:  Blah
```

Local Symbols

When symbols are exported, both the encrypted program and the unencrypted environment access the same instance. In both cases, the symbols are global. Therefore, all symbols declared local to either segment will not be seen by the other. For example:

ENCRYPT4.RLE:

```
PROC Foo
    LOCAL x
    EXPORT x
    x = 5
END PROC
Foo
```

When this encrypted program is run, the x defined in the procedure is local, and its value is lost when the procedure ends. The `EXPORT` statement is treated as if it occurred outside any procedure, and a global x is created that is visible by both the encrypted and unencrypted programs. The value of this public x is undefined because the assignment in the procedure affects the local version of x , and not the global exported version.

Clearing Exported Symbols

The effect of a `CLEAR` statement, which destroys an exported symbol, depends on where the `CLEAR` occurred. If the encrypted program clears a symbol, it becomes undefined in both the encrypted and non-encrypted segments. If the non-encrypted segment clears a symbol, it will be undefined in the non-encrypted segment, but the encrypted segment will maintain its own version.

The exception to this rule is with family members. If either segment clears a member, it will become undefined in both segments. As long as a family is exported, both segments will have access to the same members.

Appendix A

CA-Realizer Keywords

Overview

This appendix provides a list of all CA-Realizer keywords in alphabetical order. Keywords shown in *bold italic* are new in Version 2.0.

CA-Realizer Keywords

Abs	<i>Bin\$</i>
ACos	<i>BinToNum</i>
<i>AddSys</i>	<i>BitAnd</i>
AddToDate	<i>BitClear</i>
AddToTime	<i>BitLShift</i>
ALIAS	<i>BitNot</i>
And	<i>BitOr</i>
<i>AnimateCells</i>	<i>BitRShift</i>
<i>AnimateControl</i>	<i>BitSet</i>
<i>AnimateFrame</i>	<i>BitTest</i>
<i>AnimateSelect</i>	<i>BitXor</i>
<i>AnimateSpecialFrame</i>	BoardControl
<i>AppSetProc</i>	BoardNew
ARRAY	BoardQ
AS	BoardQUnique
Asc	BoardSelect
ASin	BoardSetProc
ATan2	BoardUpdate
ATn	Bool
BASE	BYTE
BEEP	CALL

<i>CASE</i>	ChartXYLine
<i>CDBL</i>	ChartXYMark
Ceil	<i>CHDIR</i>
CHAIN	Chr\$
CHAR	<i>CINT</i>
ChartBar	CLEAR
ChartControl	ClipGet
<i>ChartControlGrid</i>	ClipSet
ChartControlKey	<i>CLNG</i>
ChartControlLabels	<i>ColorNames</i>
ChartControlPane	<i>CommClose</i>
<i>ChartHBar</i>	<i>CommOpen</i>
ChartLine	<i>CommQ</i>
ChartMark	<i>CommRead</i>
ChartNew	<i>CommWrite</i>
<i>ChartPie</i>	<i>CONST</i>
ChartQ	Cos
ChartQPtInfo	<i>CSNG</i>
ChartQUnique	CUSTOMCONTROLS
<i>ChartQXAxis</i>	<i>CVD</i>
<i>ChartQYAxis</i>	<i>CVDMBF</i>
<i>ChartRemove</i>	<i>CVI</i>
ChartSelect	<i>CVL</i>
ChartSelectPane	<i>CVS</i>
ChartSetColor	<i>CVSMBF</i>
<i>ChartSetFont</i>	<i>DATE\$</i>
ChartSetKey	DateInfo
ChartSetLineStyle	<i>DATETIME</i>
ChartSetMark	DateToNum
ChartSetProc	DayOfWeek
ChartSetTitle	DDEControl
ChartSetXAxis	DDENew
ChartSetXAxis2	<i>DDEQ</i>
<i>ChartSetXGrid</i>	DDEQUnique
<i>ChartSetXLabels</i>	DDESelect
ChartSetXView	DDESetProc
ChartSetYAxis	<i>DDEWait</i>
<i>ChartSetYGrid</i>	<i>DECLARE</i>
<i>ChartSetYLabels</i>	<i>DIM</i>
ChartText	<i>DOUBLE</i>
<i>ChartUpdate</i>	<i>DWORD</i>

ELSE
ELSEIF
END
EndValid
EOF
EQV
ERASE
ERL
ERR
ERROR
EXECUTE
EXIT
Exp
Exp10
EXTERNAL
FAMILY
FileClose
FileDB
FileDelete
FileEOF
FileExport
FileImport
FileMakeName
FileOpen
FilePos
FileQ
FileQUnique
FileRead
FILES
FileSeek
FileWrite
FirstMatch
Fix
Floor
FontControl
FontNew
FontQ
FontQUnique
FontSelect
FOR
FormControl

FormModifyObject
FormNew
FormQ
FormQNum
FormQObject
FormQStr
FormQUnique
FormSelect
FormSetColor
FormSetGrid
FormSetObject
FormSetProc
FormWait
FREEFILE
FUNC
FUNCTION
GLOBAL
GOTO
Help
HelpSetProc
Hex\$
HexToNum
HIDE
IDLE
IF
IMP
in
Index
INPUT
InStr
Int
INTEGER
IS
KILL
LBound
LCase\$
Left\$
Len
LFLUSH
line
LOCAL

<i>LOF</i>	<i>MKSMBF\$</i>
Log	mm
Log10	MOD
LogControl	NEXT
LogNew	NOT
LogQ	NumToDate
<i>LogQData</i>	NumToStr
LogQSize	<i>Oct\$</i>
LogQStr	<i>OctToNum</i>
LogQUnique	OF
LogSelect	ON
<i>LogSetData</i>	<i>OPTION</i>
LogSetProc	OR
LONG	pct
LOOP	<i>PictureClipGet</i>
LPRINT	<i>PictureClipSet</i>
LROWPRINT	<i>PictureControl</i>
<i>LTrim\$</i>	<i>PictureNew</i>
MACRO	<i>PictureQ</i>
Mask	<i>PictureQUnique</i>
<i>MatInvert</i>	<i>PictureSelect</i>
<i>MatMult</i>	POINTER
<i>MatTranspose</i>	PRINT
Max	<i>PrinterControl</i>
MenuControl	PROC
<i>MenuDoCmd</i>	Product
MenuNew	PROGRAM
MenuQ	<i>PROTECT</i>
MenuQUnique	pt
MenuSelect	pxl
MenuSetCmd	QDate
MenuSetProc	<i>QDir</i>
<i>MessageBox</i>	QFormat
Mid\$	QNOptMods
Min	QNOptParams
<i>MKD\$</i>	QOptMod
<i>MKDIR</i>	QOptParam
<i>MKDMBF\$</i>	QSys
<i>MKI\$</i>	QVar
<i>MKL\$</i>	RANDOMIZE
<i>MKS\$</i>	REAL

RENAME	SheetUpdate
RESET	SHELL
<i>ResetHourglass</i>	Sin
RESUME	SINGLE
RETURN	SLEEP
<i>Reverse\$</i>	<i>Space\$</i>
Right\$	<i>Spc</i>
RMDIR	Sprint
Rnd	Sqr
Round	Sqrt
ROWPRINT	StartValid
<i>RTrim\$</i>	STATIC
RUN	StatMovAvg
RunTot	StatMovMax
SchedControl	StatMovMin
SchedNew	StatMovTot
SchedQ	StatNorm
<i>SchedQData</i>	StatPack
SchedRemove	StatPercent
SchedSet	StatRank
<i>ScreenEGA</i>	StatRegress
SEEK	StatSort
SELECT	StatSwitch
<i>SetAppName</i>	StatUnpack
<i>SetDir</i>	<i>StdColor</i>
SetFormat	StdFont
<i>SetHourglass</i>	StdOpen
SetSys	StdPrintConfig
SetSystemDate	StdSaveAs
SetSystemTime	STEP
Sgn	STOP
SheetControl	<i>StrDelete\$</i>
<i>SheetControlLabels</i>	STRING
SheetNew	String\$
SheetQ	<i>StrInsert\$</i>
<i>SheetQInfo</i>	<i>StrKeyword</i>
SheetQUnique	StrToDate
SheetSelect	<i>StrTok</i>
<i>SheetSetColLabels</i>	StrToNum
SheetSetProc	STRUCTURE
<i>SheetSetRowLabels</i>	SUB

SubStr\$	VectSet
Sum	WEND
SWAP	WHILE
SYSTEM	WORD
<i>TabletArc</i>	XOR
<i>TabletBitmap</i>	ZeroFill
<i>TabletBrush</i>	
<i>TabletChord</i>	
<i>TabletControl</i>	
<i>TabletEllipse</i>	
<i>TabletGetPixel</i>	
<i>TabletLine</i>	
<i>TabletLineTo</i>	
<i>TabletMouseX</i>	
<i>TabletMouseY</i>	
<i>TabletMoveTo</i>	
<i>TabletPen</i>	
<i>TabletPie</i>	
<i>TabletPolygon</i>	
<i>TabletPolyline</i>	
<i>TabletPolyPolygon</i>	
<i>TabletRectangle</i>	
<i>TabletRoundRect</i>	
<i>TabletSelect</i>	
<i>TabletSetColor</i>	
<i>TabletSetPixel</i>	
<i>TabletText</i>	
<i>TabletTextColor</i>	
<i>TabletUndo</i>	
Tan	
THEN	
<i>TIMES</i>	
<i>TIMER</i>	
TO	
<i>twip</i>	
TYPE	
<i>UBound</i>	
<i>UCase\$</i>	
UNTIL	
USING	
Val	

Appendix B

CA-Realizer Predefined Constants

Overview

This appendix provides a list of all CA-Realizer constants in alphabetical order. Constants shown in ***bold italic*** are new in Version 2.0.

Constants

<i>_Abort</i>	<i>_ArrowSW</i>
<i>_AccelNotify</i>	<i>_ArrowW</i>
<i>_Advise</i>	<i>_Background</i>
<i>_All</i>	<i>_Bar</i>
<i>_AllData</i>	<i>_Beg</i>
<i>_AllFormats</i>	<i>_Binary</i>
<i>_AllText</i>	<i>_Bitmap</i>
<i>_AllTitle</i>	<i>_BitmapButton</i>
<i>_AllYAxis</i>	<i>_Black</i>
<i>_Alpha</i>	<i>_Blue</i>
<i>_AlphaFormat</i>	<i>_BlueGreen</i>
<i>_Alt</i>	<i>_Board</i>
<i>_Animate</i>	<i>_Bold</i>
<i>_Append</i>	<i>_Borderless</i>
<i>_Array</i>	<i>_Bottom</i>
<i>_ArrowE</i>	<i>_BottomTitle</i>
<i>_ArrowN</i>	<i>_Brick</i>
<i>_ArrowNE</i>	<i>_BringToFront</i>
<i>_ArrowNW</i>	<i>_Brown</i>
<i>_ArrowS</i>	<i>_Brush</i>
<i>_ArrowSE</i>	<i>_BufferedMetafile</i>

<u>Button</u>	<u>CQ_Parity</u>
<u>Cancel</u>	<u>CQ_ReadBuffSize</u>
<u>Candlestick</u>	<u>CQ_ReadPending</u>
<u>CaptionCenter</u>	<u>CQ_StopBits</u>
<u>CaptionLeft</u>	<u>CQ_WriteBufSize</u>
<u>CaptionRight</u>	<u>CQ_WritePending</u>
<u>CaseSensitive</u>	<u>CQ_XON</u>
<u>CCM</u>	<u>Cream</u>
<u>CCStyle</u>	<u>CTPos_Bottom</u>
<u>Center</u>	<u>CTPos_Center</u>
<u>CenturySplit</u>	<u>CTPos_Left</u>
<u>Change</u>	<u>CTPos_Right</u>
<u>ChangeDrive</u>	<u>CTPos_Top</u>
<u>Char</u>	<u>Cur</u>
<u>Chart</u>	<u>CustomMessage</u>
<u>ChartBackgr</u>	<u>Cyan</u>
<u>ChartText</u>	<u>DarkBrown</u>
<u>Check</u>	<u>DarkGreen</u>
<u>CheckBitmap</u>	<u>DarkGreenBlue</u>
<u>CheckBox</u>	<u>DarkPurple</u>
<u>Circle</u>	<u>Dash</u>
<u>Clear</u>	<u>DashDot</u>
<u>Click</u>	<u>DashDotDot</u>
<u>Close</u>	<u>Data</u>
<u>CmdLine</u>	<u>Data1</u>
<u>Col1</u>	<u>Data2</u>
<u>Col2</u>	<u>Data3</u>
<u>ColButton</u>	<u>DateFormat</u>
<u>Color</u>	<u>DateNum</u>
<u>ColRange</u>	<u>DateTime</u>
<u>ComboBox</u>	<u>DBF</u>
<u>ComboDrop</u>	<u>DDE</u>
<u>ComboList</u>	<u>Default</u>
<u>Comm</u>	<u>DefButton</u>
<u>Compete</u>	<u>Defined</u>
<u>ContextEnter</u>	<u>DefLB</u>
<u>Control</u>	<u>DI_DateNum</u>
<u>Copies</u>	<u>DI_DayOfMonth</u>
<u>CQ_BaudRate</u>	<u>DI_DayOfWeek</u>
<u>CQ_ByteSize</u>	<u>DI_DayOfYear</u>
<u>CQ_CID</u>	<u>DI_Hour</u>

<u>_DI_Minute</u>	<u>_FormNum</u>
<u>_DI_Month</u>	<u>_FPErrors</u>
<u>_DI_Second</u>	<u>_FQO_Height</u>
<u>_DI_TimeNum</u>	<u>_FQO_HWnd</u>
<u>_DI_Year</u>	<u>_FQO_ItemList</u>
<u>_Diamond</u>	<u>_FQO_ItemNum</u>
<u>_Directory</u>	<u>_FQO_ItemType</u>
<u>_DOS</u>	<u>_FQO_Left</u>
<u>_Dot</u>	<u>_FQO_Selected</u>
<u>_Down</u>	<u>_FQO_Style</u>
<u>_Drag</u>	<u>_FQO_Top</u>
<u>_DragNDrop</u>	<u>_FQO_Value</u>
<u>_DragOff</u>	<u>_FQO_Width</u>
<u>_Drives</u>	<u>_Frame</u>
<u>_Drop</u>	<u>_Gray</u>
<u>_DropDownCombo</u>	<u>_GrayGreen</u>
<u>_DropDownList</u>	<u>_Green</u>
<u>_DropOff</u>	<u>_GreenYellow</u>
<u>_EditMenu</u>	<u>_Grid</u>
<u>_EditText</u>	<u>_GroupBox</u>
<u>_Empty</u>	<u>_Heavy</u>
<u>_End</u>	<u>_Help</u>
<u>_ErrorLog</u>	<u>_HelpIndex</u>
<u>_EvenParity</u>	<u>_HelpMenu</u>
<u>_Evergreen</u>	<u>_Hidden</u>
<u>_Excel</u>	<u>_Hide</u>
<u>_Execute</u>	<u>_Horiz</u>
<u>_Exists</u>	<u>_HotClick</u>
<u>_Export</u>	<u>_HPxIPerInch</u>
<u>_Extended</u>	<u>_Hwnd</u>
<u>_Family</u>	<u>_IBeam</u>
<u>_FamilyColwise</u>	<u>_Icon</u>
<u>_FamilyRowwise</u>	<u>_Ignore</u>
<u>_Field</u>	<u>_Import</u>
<u>_File</u>	<u>_Inactive</u>
<u>_FileMenu</u>	<u>_Info</u>
<u>_Fill</u>	<u>_Initiate</u>
<u>_FillForm</u>	<u>_Inside</u>
<u>_Font</u>	<u>_Integer</u>
<u>_Form</u>	<u>_IntegerFormat</u>
<u>_Format</u>	<u>_Invoke</u>

<u>Italics</u>	<u>Log_NumVisLines</u>
<u>ItemNum</u>	<u>Log_ScrollLine</u>
<u>ItemType</u>	<u>Log_ScrollTo</u>
<u>KeyBackgr</u>	<u>Log_Selection</u>
<u>KeyBorder</u>	<u>Log_SelectLine</u>
<u>KeyLabels</u>	<u>Log_SelectStr</u>
<u>KeyText</u>	<u>Log_Str</u>
<u>KeyTitle</u>	<u>Log_StrSize</u>
<u>Labels</u>	<u>Lotus</u>
<u>Landscape</u>	<u>MacroDir</u>
<u>Left</u>	<u>Magenta</u>
<u>LeftTitle</u>	<u>MarkDelims</u>
<u>Light</u>	<u>Marker</u>
<u>LightEvergreen</u>	<u>MarkParity</u>
<u>LightGray</u>	<u>Maroon</u>
<u>LightGreen</u>	<u>Mask</u>
<u>LightYellow</u>	<u>Maximize</u>
<u>LimeGreen</u>	<u>MB_AbortRetryIgnore</u>
<u>Line</u>	<u>MB_Asterisk</u>
<u>ListBox</u>	<u>MB_Exclamation</u>
<u>ListDelete</u>	<u>MB_Hand</u>
<u>ListFams</u>	<u>MB_Information</u>
<u>ListFiles</u>	<u>MB_Ok</u>
<u>ListFonts</u>	<u>MB_OkCancel</u>
<u>ListFontSizes</u>	<u>MB_Question</u>
<u>ListInsert</u>	<u>MB_RetryCancel</u>
<u>ListPrFonts</u>	<u>MB_Stop</u>
<u>ListPrFontSizes</u>	<u>MB_YesNo</u>
<u>ListSelect</u>	<u>MB_YesNoCancel</u>
<u>ListVars</u>	<u>Medium</u>
<u>LoadDir</u>	<u>MediumBlue</u>
<u>Log</u>	<u>MediumBrown</u>
<u>Log_CharLine</u>	<u>MediumGreen</u>
<u>Log_DeleteLine</u>	<u>Menu</u>
<u>Log_DeleteSelection</u>	<u>MenuNum</u>
<u>Log_InsertLine</u>	<u>Metafile</u>
<u>Log_LimitText</u>	<u>MiddleClick</u>
<u>Log_LineChar</u>	<u>Minimize</u>
<u>Log_LineLen</u>	<u>Move</u>
<u>Log_LineStr</u>	<u>MultiLine</u>
<u>Log_NumLines</u>	<u>MultipleSel</u>

<u>Name</u>	<u>Peek</u>
<u>Named</u>	<u>Pen</u>
<u>NavyBlue</u>	<u>PI</u>
<u>No</u>	<u>Pick</u>
<u>NoBorder</u>	<u>PickDrag</u>
<u>NoColHeaders</u>	<u>Pink</u>
<u>NoFamilyHeaders</u>	<u>Plain</u>
<u>NoFieldHeaders</u>	<u>Play</u>
<u>NoFiles</u>	<u>Plus</u>
<u>NoFooter</u>	<u>Point</u>
<u>NoFrame</u>	<u>Poke</u>
<u>NoHeader</u>	<u>Popup</u>
<u>NoHLines</u>	<u>Portrait</u>
<u>NoLabels</u>	<u>PowderBlue</u>
<u>NoMax</u>	<u>Print</u>
<u>None</u>	<u>PrintBest</u>
<u>NonMDI</u>	<u>Printer</u>
<u>NoParity</u>	<u>PrintLog</u>
<u>Normal</u>	<u>Proc</u>
<u>NormFiles</u>	<u>ProgDir</u>
<u>NoRowHeaders</u>	<u>Purple</u>
<u>NotExtended</u>	<u>RadioButton</u>
<u>Notify</u>	<u>Range</u>
<u>NoVLines</u>	<u>Read</u>
<u>Number0</u>	<u>ReadOnly</u>
<u>Number1</u>	<u>ReadWrite</u>
<u>Number2</u>	<u>Real</u>
<u>NumDims</u>	<u>RealFormat</u>
<u>OddParity</u>	<u>Realizer</u>
<u>Off</u>	<u>Record</u>
<u>OK</u>	<u>Rectangle</u>
<u>OLE</u>	<u>RectCross</u>
<u>OptionBitmap</u>	<u>Red</u>
<u>OptionButton</u>	<u>Redraw</u>
<u>Orange</u>	<u>Remove</u>
<u>OrigDir</u>	<u>Replace</u>
<u>OutputWidth</u>	<u>Request</u>
<u>Pale</u>	<u>Reset</u>
<u>PaneBackgr</u>	<u>Restore</u>
<u>Pause</u>	<u>Retry</u>
<u>PCX</u>	<u>Right</u>

<u>RightClick</u>	<u>RLZM_PASTE</u>
<u>RightSideT</u>	<u>RLZM_PRINT</u>
<u>RightTitle</u>	<u>RLZM_PRINTSETUP</u>
<u>RLZM_ABOUT</u>	<u>RLZM_REPLACE</u>
<u>RLZM_AGAIN</u>	<u>RLZM_RESET</u>
<u>RLZM_ARRANGE</u>	<u>RLZM_RESETPROG</u>
<u>RLZM_BACKSP</u>	<u>RLZM_RETURN</u>
<u>RLZM_BALANCE</u>	<u>RLZM_RUNLINE</u>
<u>RLZM_CALL_TREE</u>	<u>RLZM_RUNPROG</u>
<u>RLZM_CASCADE</u>	<u>RLZM_RUNSEL</u>
<u>RLZM_CLEAR</u>	<u>RLZM_SAVE</u>
<u>RLZM_CLEAR2</u>	<u>RLZM_SAVEAS</u>
<u>RLZM_CLOSE</u>	<u>RLZM_SHOW_SYMB</u>
<u>RLZM_COPY</u>	<u>RLZM_SHOWDIRS</u>
<u>RLZM_COPYWINDOW</u>	<u>RLZM_SHOWERR</u>
<u>RLZM_COPYWINDOW1</u>	<u>RLZM_SHOWERR1</u>
<u>RLZM_COPYWINDOW2</u>	<u>RLZM_SHOWERR2</u>
<u>RLZM_CUT</u>	<u>RLZM_SHOWLOG</u>
<u>RLZM_DDE_LOG</u>	<u>RLZM_SHOWLOG1</u>
<u>RLZM_DDE_LOG_OFF</u>	<u>RLZM_SHOWLOG2</u>
<u>RLZM_DDE_LOG_ON</u>	<u>RLZM_SHOWSCHED</u>
<u>RLZM_DEBUG_OFF</u>	<u>RLZM_SKIP</u>
<u>RLZM_DEBUG_ON</u>	<u>RLZM_STEP</u>
<u>RLZM_DEBUG_TOG</u>	<u>RLZM_STOP</u>
<u>RLZM_EXECUTE</u>	<u>RLZM_STOP1</u>
<u>RLZM_EXIT</u>	<u>RLZM_STOP2</u>
<u>RLZM_FIND</u>	<u>RLZM_TILE</u>
<u>RLZM_FULLSTOP</u>	<u>RLZM_TILE1</u>
<u>RLZM_GOTO</u>	<u>RLZM_TILE2</u>
<u>RLZM_HELP</u>	<u>RLZM_TRACE</u>
<u>RLZM_HELP_COMMAND</u>	<u>RLZM_UNDO</u>
<u>RLZM_HELP_HELP</u>	<u>RLZM_WINDOW_1</u>
<u>RLZM_HELP_INDEX</u>	<u>RLZM_WINDOW_2</u>
<u>RLZM_HELP_KEYBOARD</u>	<u>RLZM_WINDOW_3</u>
<u>RLZM_HELP_LANGUAGE</u>	<u>RLZM_WINDOW_4</u>
<u>RLZM_HELP_PROCEDURE</u>	<u>RLZM_WINDOW_5</u>
<u>RLZM_KEYSAVE</u>	<u>RLZM_WINDOW_6</u>
<u>RLZM_MOREWINDS</u>	<u>RLZM_WINDOW_7</u>
<u>RLZM_NEW</u>	<u>RLZM_WINDOW_8</u>
<u>RLZM_NOTHING</u>	<u>RLZM_WINDOW_9</u>
<u>RLZM_OPEN</u>	<u>Row</u>

_RowRange
_RunMenu
_Scalar
_Sched
_Screen
_Scroll
_Select
_Selected
_Selection
_SelectOff
_SendBehind
_Separator
_SetColor
_SetDrag
_SetFocus
_SetFont
_SetFooter
_SetHeader
_SetMode
_SetOffset
_SetRedraw
_SetSpeed
_SetTabs
_Sheet
_Shift
_Show
_Size
_SizeInside
_SizeMDI
_SizeTrack
_Solid
_Sorted
_SpaceParity
_Stagger
_Star
_Start
_StartAtNext
_Stop
_Strikeout
_Style
_SubDirs

_SuperCalc
_SysVar
_Tablet
_Tan
_Text
_TextBox
_TimeNum
_Title
_Toolbar
_Top
_ToPlot
_TopTitle
_ToValue
_Triangle
_Turquoise
_Unadvise
_Uncheck
_Underline
_Undo
_Up
_Update
_UpsideT
_UseRealizer
_Var
_Version
_Vert
_VertBar
_VerticalLine
_Violet
_Virtual
_VK_ADD
_VK_BACK
_VK_CAPITAL
_VK_CLEAR
_VK_CONTROL
_VK_DECIMAL
_VK_DELETE
_VK_DIVIDE
_VK_DOWN
_VK_END
_VK_ESCAPE

<code>_VK_EXECUTE</code>	<code>_VK_SPACE</code>
<code>_VK_F1</code>	<code>_VK_SUBTRACT</code>
<code>_VK_F10</code>	<code>_VK_TAB</code>
<code>_VK_F11</code>	<code>_VK_UP</code>
<code>_VK_F12</code>	<code>_VPxlPerInch</code>
<code>_VK_F2</code>	<code>_White</code>
<code>_VK_F3</code>	<code>_WindowMenu</code>
<code>_VK_F4</code>	<code>_Write</code>
<code>_VK_F5</code>	<code>_XAxis</code>
<code>_VK_F6</code>	<code>_XAxis_Color</code>
<code>_VK_F7</code>	<code>_XAxis2</code>
<code>_VK_F8</code>	<code>_XMark</code>
<code>_VK_F9</code>	<code>_XPos</code>
<code>_VK_HELP</code>	<code>_YAxis</code>
<code>_VK_HOME</code>	<code>_YAxis_Color</code>
<code>_VK_INSERT</code>	<code>_Yellow</code>
<code>_VK_LEFT</code>	<code>_Yes</code>
<code>_VK_MENU</code>	<code>_YPos</code>
<code>_VK_MULTIPLY</code>	
<code>_VK_NEXT</code>	
<code>_VK_NUMLOCK</code>	
<code>_VK_NUMPAD0</code>	
<code>_VK_NUMPAD1</code>	
<code>_VK_NUMPAD2</code>	
<code>_VK_NUMPAD3</code>	
<code>_VK_NUMPAD4</code>	
<code>_VK_NUMPAD5</code>	
<code>_VK_NUMPAD6</code>	
<code>_VK_NUMPAD7</code>	
<code>_VK_NUMPAD8</code>	
<code>_VK_NUMPAD9</code>	
<code>_VK_PAUSE</code>	
<code>_VK_PRINT</code>	
<code>_VK_PRIOR</code>	
<code>_VK_RETURN</code>	
<code>_VK_RIGHT</code>	
<code>_VK_SCROLL</code>	
<code>_VK_SELECT</code>	
<code>_VK_SEPARATOR</code>	
<code>_VK_SHIFT</code>	
<code>_VK_SNAPSHOT</code>	

Appendix C

The ASCII Character Set

This appendix provides technical information about the ASCII character set.

Dec	Hex	Character
-----	-----	-----------

0	00	NUL (Null)
1	01	SOH (Start of heading)
2	02	STX (Start of text)
3	03	ETX (End of text)
4	04	EOT (End of trans)
5	05	ENQ (Enquiry)
6	06	ACK (Acknowledge)
7	07	BEL (Bell)
8	08	BS (Backspace)
9	09	HT (Horizontal tab)
10	0A	LF (Linefeed)
11	0B	VT (Vertical tab)
12	0C	FF (Formfeed)
13	0D	CR (Carriage return)
14	0E	SO (Shift out)
15	0F	SI (Shift in)
16	10	DLE (Data link escape)
17	11	DC1 (Device control 1)
18	12	DC2 (Device control 2)
19	13	DC3 (Device control 3)
20	14	DC4 (Device control 4)
21	15	NAK (Negative acknowledge)

Dec	Hex	Character
-----	-----	-----------

22	16	SYN (Synchronous idle)
23	17	ETB (End trans block)
24	18	CAN (Cancel)
25	19	EM (End of medium)
26	1A	SUB (Substitute)
27	1B	ESC (Escape)
28	1C	FS (File separator)
29	1D	GS (Group separator)
30	1E	RS (Record separator)
31	1F	US (Unit separator)
32	20	(Space)
33	21	!
34	22	"
35	23	#
36	24	\$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+

Dec	Hex	Character	Dec	Hex	Character	Dec	Hex	Character
44	2C	,	75	4B	K	106	6A	j
45	2D	-	76	4C	L	107	6B	k
46	2E	.	77	4D	M	108	6C	l
47	2F	/	78	4E	N	109	6D	m
48	30	0	79	4F	O	110	6E	n
49	31	1	80	50	P	111	6F	o
50	32	2	81	51	Q	112	70	p
51	33	3	82	52	R	113	71	q
52	34	4	83	53	S	114	72	r
53	35	5	84	54	T	115	73	s
54	36	6	85	55	U	116	74	t
55	37	7	86	56	V	117	75	u
56	38	8	87	57	W	118	76	v
57	39	9	88	58	X	119	77	w
58	3A	:	89	59	Y	120	78	x
59	3B	;	90	5A	Z	121	79	y
60	3C	<	91	5B	[	122	7A	z
61	3D	=	92	5C	\	123	7B	{
62	3E	>	93	5D	]	124	7C	
63	3F	?	94	5E	^	125	7D	}
64	40	@	95	5F	_	126	7E	~
65	41	A	96	60	`	127	7F	DEL (Delete)
66	42	B	97	61	a			
67	43	C	98	62	b			
68	44	D	99	63	c			
69	45	E	100	64	d			
70	46	F	101	65	e			
71	47	G	102	66	f			
72	48	H	103	67	g			
73	49	I	104	68	h			
74	4A	J	105	69	i			

Index

#

#INF, 3-30

#QIND, 3-30

#QNAN, 3-30

&

& character as a format code, 4-21

_

_AlphaFormat, 21-11

_CaseSensitive, 3-5, 21-12

_CenturySplit, 21-12

_CmdLine, 21-14

_Color, 21-13

_ControlHeld, 9-24

_CustomLP, 9-29

_CustomWP, 9-29

_DateFormat, 7-33, 21-11

_DateNum, 3-5, 3-33

_DateTime, 7-18

_DragForm, 9-29

_DragItem, 9-29

_ErrorLog, 21-9

_EvenParity, 24-2

_FPErrors, 21-13

_Hwnd, 21-14

_ItemType, 9-28

_LoadDir, 21-7

_MacroDir, 21-7

_MarkParity, 24-2

_mn alias, 6-20

_NoParity, 24-2

_OddParity, 24-2

_OpSys, 21-14

_OrigDir, 21-7

_OutputWidth, 21-11

_pn alias, 6-20

_PrintBest, 21-14

_PrintLog, 21-9

_ProgDir, 21-9

_Real, 7-18

_RealFormat, 7-33, 21-11

_Separator, 21-11

_ShiftHeld, 9-25

_Size, 21-5

_SizeInside, 21-6

_SizeMDI, 21-7

- _SpaceParity, 24-2
- _TimeNum, 3-5, 3-33
- _UseRealizer, 9-29
- _Version, 21-13
- _XPos, 9-25
- _YPos, 9-25

A

Abs, 3-2, 3-17

Accelerator key

- ALT+char combination in a form, 15-23
- specifying
 - a single key accelerator for a menu item, 18-22
 - an ALT key combination for a menu item, 18-21
 - for a menu, 18-4
 - for form objects, see Form

ACos, 3-15

Actual parameter

- defined, 6-9 to 6-10
- rules for usage, 6-9 to 6-10

Adding

- data to a Board, see Boardwatch
- form interface tools, see Form
- separators, 18-11

AddSys, 7-6, 21-7

AddToDate, 3-31, 3-35

AddToTime, 3-31, 3-35, 3-36

Adjusting valid ranges of arrays, 3-13

Advanced data types, 2-3, 2-5

- arrays of records, 2-48
- dynamic arrays, 2-26
- families, 2-11
- records, 2-45
- static arrays, 2-26
- structures, 2-45

ALIAS, 26-3, 26-10

Alpha data type, 2-2, 2-16

And, 3-6

AnimateCells, 10-6, 17-4

AnimateControl, 17-6 to 17-12, 17-8

AnimateFrame, 17-6

Animation, see Animator

Animator

- animation sequence, 17-3
- commands, 17-1
- controlling, 17-6 to 17-12
- defining
 - a sequence of animation cells, 17-4
 - an animation frame, 17-6
- frames, 17-2
- functions, 17-1
- how it works, 17-2
- mouse click processing, 17-8 to 17-9
- moving the window, 17-8
- notification, 17-8 to 17-9
- offset position, 17-8
- pause frame, 17-8
- restarting, 17-7
- special frames, 17-8
- speed factor, 17-8
- starting, 17-7
- stop frame, 17-8
- stopping, 17-7
- using pictures, 17-9 to 17-12

Application

- name, specifying in DDE, 25-3
- procedure, 9-32

AppSetProc, 9-32

Arrays

- accessing elements using the index number, 2-26
- adjusting valid ranges, 3-13
- applying comparison and logical operators to, 3-12
- CA-Realizer enhancements to conventional BASIC, 2-29
- changing bounds dynamically, 2-33
- constructing, 2-27 to 2-29
- containing random numbers, 3-19
- creating
 - general form, 2-27
 - using the Index function, 2-30, 3-21
 - using VectSet, 3-21
- definition of, 2-10
- determining
 - the moving average of, 3-28
 - the valid range of, 3-22
- dynamic expansion of, 2-33
- elements of, 2-25
- explicit
 - declaration with DIM, 2-33
 - redimensioning, 2-33
- extracting an element, 2-30
- features unique to CA-Realizer, 2-29 to 2-37
- functions, 3-14 to 3-23
- inserting an element, 2-32
- limit on dimensions of, 2-26
- limiting size, 2-42 to 2-44
- maximum number of elements, 2-42 to 2-44
- morphing, 2-35 to 2-37
- multi-dimensional, 2-26
- normalizing elements using StatNorm, 3-25
- of arrays, 2-39 to 2-42
- of families, 2-53
- of records, 2-48
- packing and switching elements in, 3-26

processing

- in CA-Realizer, 3-11 to 3-29
- in conventional BASIC, 3-11
- mixing scalars and arrays in, 3-13
- using the shift operators, 3-13
- querying number of dimensions, 2-18
- ranking elements using StatRank, 3-25
- reducing an element, 2-31
- removing an element, 2-31
- rules
 - for computation, 3-13
 - for operating on, 2-37 to 2-44
 - for printing, 2-44
- shifting elements in, 3-13
- sorting elements using StatSort, 3-25
- subarrays, 2-39 to 2-42, 2-44
- subranging, 2-30
- subscripting elements of, 3-14
- summing elements using SUM, 3-21
- transposing, 2-32, 2-36
- valid range of, 2-27
- working with array values, 3-11 to 3-29
- writing strings to a file, 7-14

Asc, 4-2

ASCII

- codes, 4-1
- converting to text, 4-2 to 4-4

Asin, 3-15

Assigning

- unique file numbers, 7-4
- values
 - to a parameter in an expression, 6-11
 - to a variable, 2-14

Assignment operator, 2-14

Asynchronous tool procedures, 9-30

Atn, 3-15

B

BASIC, differences in file management,
7-1 to 7-3

Batch file creation, 22-4 to 22-5

Bin\$, 4-16

Binary

- files, 7-1
- formats, 7-34
- notation, 2-6

BinToNum, 4-18

BitAnd, 3-8

BitClear, 3-8

BitLShift, 3-8

Bitmap graphics, 16-1

BitNot, 3-8

BitOr, 3-8

BitRShift, 3-8

BitSet, 3-8

BitTest, 3-8

Bitwise Operators, 3-7 to 3-8

BitXor, 3-8

board procedure, attaching, 13-13

BoardControl, 13-3

BoardNew, 13-3

BoardQUnique, 13-4

Boards, see Boardwatch

BoardSetProc, 13-13

BoardUpdate, 13-3, 13-7, 13-10

Boardwatch, 13-1 to 13-15

adding

- a variable to, 13-7, 13-12
- data, 13-10

attaching a board procedure, 13-13

automatically recomputing the size,
13-3, 13-11

changing

- column format, 13-8
- column titles, 13-7

clearing a variable, 13-12

command summary, 13-1

creating, 13-1 to 13-10

customizing with a board procedure,
13-13

data, 13-1 to 13-3

default formats for, 13-8

listing member names, 13-6

overview, 13-1 to 13-3

processing in the board procedure,
13-14

specifying

- a template, 13-5 to 13-10
- the layout, 13-9

transposing elements, 13-9

updating, 13-7

Boolean array, using with StatPack, 3-26

Breaking a long statement, 5-2

C

CA-Compete! files

- exporting, 7-31
- importing, 7-31

CA-Realizer

data types in, 2-1 to 2-24

file concepts, 7-4

file management compared to earlier
BASICS, 7-1 to 7-3

functions, 3-10 to 3-36

menus, 18-8

- notation
 - binary, 2-6
 - hexadecimal, 2-6
 - octal, 2-6
- operators, 3-1 to 3-8
- types of iterative statements, 5-10
- unique array-handling features, 2-29 to 2-37
- CA-SuperCalc files
 - exporting, 7-31
 - importing, 7-31
- CA-RET
 - executing a CARET command, 25-18
 - initiating conversation, 25-18
 - requesting information from, 25-19
 - terminating conversation, 25-19
 - using with CA-Realizer, 25-17 to 25-19
- Calling
 - a function recursively, 6-21 to 6-22
 - a procedure recursively, 6-21 to 6-22
 - procedures and functions, 6-2
- CASE, see SELECT
- Case sensitivity, 3-5, 4-14, 21-12
- Ceil, 3-18
- Changing
 - a font using the StdFont dialog box, 8-8
 - default PRINT formats using the SetSys command, 8-28
 - directories, 7-7
 - information in a log, 14-10
 - printer configuration, 8-6
 - variable type, 2-17
- Chart procedure, see Charts
- ChartBar, 11-6, 11-8
- ChartControl, 11-6, 11-36, 11-57
- ChartControlGrid, 11-43
- ChartControlKey, 11-33, 11-36
- ChartControlLabels, 11-39
- ChartControlPane, 11-51
- ChartLine, 11-12 to 11-15, 11-36
- ChartMark, 11-12 to 11-15, 11-36
- ChartNew, 9-6
- ChartPie, 11-17
- ChartQ, 9-8 to 9-11
- ChartQUnique, 9-4
- ChartQXAxis, 11-49
- ChartQYaxis, 11-49
- ChartRemove, 11-42, 11-49
- Charts
 - adding
 - a key to, 11-32 to 11-37
 - a title, 11-37
 - elements to a plot, 11-48
 - anatomy, 11-4
 - applying labels to, 11-37 to 11-42
 - automatic scaling of the x- and y-axis, 11-26
 - bar chart, 11-7 to 11-12
 - candlestick chart, 11-8
 - changing
 - axis labels, 11-38
 - line colors, 11-19
 - clearing, 11-51
 - commands and functions, 11-1
 - controlling
 - the appearance of, 11-18 to 11-47
 - the grid, 11-43
 - creating, 11-1 to 11-18
 - customizing with a chart procedure, 11-53
 - date-time correlation, 11-27
 - filled area chart, 11-13
 - Gantt chart, 11-11
 - hiding labels, 11-39
 - high-low chart, 11-8
 - horizontal bar chart, 11-11
 - I-beam chart, 11-8

-
- labeling
 - data points, 11-40
 - the x- and y-axis, 11-37
 - line chart, 11-12 to 11-15
 - marker chart, 11-12 to 11-15
 - modifying using Boolean array, 11-12
 - panes, 11-49 to 11-51
 - printing, 11-57
 - procedure, 9-21, 11-52 to 11-55
 - processing in the chart procedure, 11-54
 - redraw control, 11-52
 - removing a plot, 11-49
 - scroll bar, 11-31
 - secondary x-axis, 11-30
 - selecting a pane, 11-50
 - setting
 - colors, 11-19
 - fonts, 11-45
 - the chart axes, 11-26 to 11-32
 - the line style, 11-24
 - x-axis view, 11-31
 - sizing a pane, 11-51
 - updating a plot, 11-48
 - width of a line, 11-26
 - x-axis, 11-27
 - xy chart, 11-16
 - y-axis, 11-32
 - zooming the x-axis, 11-28
- ChartSelectPane, 11-50
- ChartSetColor, 11-19, 11-36
- ChartSetFont, 11-46
- ChartSetKey, 11-33
- ChartSetLineStyle, 11-24
- ChartSetMark, 11-15
- ChartSetProc, 11-53
- ChartSetTitle, 11-37
- ChartSetXAxis, 11-12, 11-27
- ChartSetXAxis2, 11-30
- ChartSetXGrid, 11-43
- ChartSetXLabels, 11-38
- ChartSetXView, 11-31
- ChartSetYAxis, 11-32, 11-50
- ChartSetYGrid, 11-43
- ChartSetYLabels, 11-38
- ChartText, 11-40
- ChartUpdate, 11-48
- ChartXYLine, 11-16
- ChartXYMark, 11-16
- CHDIR, 7-7
- Chr\$, 4-2
- CLEAR
 - used in a boardwatch, 13-12
 - used to clear variables, 2-19, 2-22
 - used to redefine a procedure or function, 6-21
- Clearing
 - logs, 14-5
 - variables, 2-19
- Clipboard, 23-1 to 23-2
 - pasting
 - a picture, 23-2
 - text, 23-1
 - retrieving
 - a picture, 23-2
 - text, 23-2
- ClipGet, 23-2
- ClipSet, 23-1
- Closing a file, see Low-level file commands
- Color
 - querying individual pixels in the graphics tablet, 16-14
 - setting
 - in a chart, 11-19
 - in a form, 15-18 to 15-20
 - in a spreadsheet, 12-7
- Column buttons, in a spreadsheet, 12-10
-

Command

- line arguments, 21-14
- statements, 5-1

CommClose, 24-6

Comment statement, 5-2

CommOpen, 24-1 to 24-3

CommQ, 24-4 to 24-6

CommRead, 24-3

Communications, see Serial communications

CommWrite, 24-3 to 24-4

Comparing strings, 4-4 to 4-5

Comparison operators, 3-4 to 3-5

Concatenating

- strings, 4-4
- using operators, 3-4

Conditional statements, 5-4 to 5-9

- IF, 5-4 to 5-7
- ON ERROR, 5-4
- SELECT, 5-4, 5-7 to 5-9
- using IF as a function, 5-7

CONST, 2-20

Constants

- declaring, 2-20
- definition of, 2-13, 2-20
- predefined, 2-20
- symbolic, 2-20
- system, 21-13 to 21-14

Context-sensitive help, 20-1

Control structures

- controlling program flow with, 5-2
- introduction to, 5-2
- nesting, 5-17

Controlling

appearance

- of a chart, 11-19
- of a form, 15-14
- of a Log, 14-5
- of a sheet, 12-6

log attributes, 14-5

program flow, 5-2

- EXIT, 5-9
- exiting an iterative statement, 5-14
- FOR...NEXT, 5-11
- IF, 5-4
- IF function, 5-7
- IF...ELSE, 5-5
- IF...ELSEIF, 5-6
- IF...THEN, 5-4
- iterative statement for, 5-10
- LOOP...END LOOP, 5-13
- ON ERROR, 5-4, 5-18
- SELECT, 5-4
- using a conditional statement for, 5-4
- using an iterative statement for, 5-4
- WHILE...END WHILE, 5-13
- with GOTO label, 5-9
- with nested IF statements, 5-6

Converting

- ASCII and Text, 4-2 to 4-4
- date-time values, 3-31
- screen-relative position to data-relative position, 11-54
- strings
 - to a real number, 4-17
 - to date-time, 4-17

Copying

- a picture from the Clipboard, 23-2
- text from the Clipboard, 23-2

Cos, 3-15

Creating

- a new spreadsheet, 12-3
- a picture, 10-6
- an array, 2-27
- boards, 13-3

- charts, 11-5
- date-time values, 3-31
- directories, 7-7

CUSTCTRL.H, 27-10

Custom control, 27-1 to 27-21

- _CCM, 27-18
- _CCSTYLE, 27-20
- adding to a form, 27-3 to 27-10
- CA-Realizer

- messages, 27-11 to 27-18
 - numeric value messages, 27-13 to 27-14
 - string value messages, 27-15 to 27-17

- creating, 27-10 to 27-21

- custom message, 27-18 to 27-19

- declaring, 27-2 to 27-3

- form colors, 27-20 to 27-21

- FormDev, 27-5

- invoking the form procedure, 27-19 to 27-20

- message order, 27-17

- messages

- CCM_CUSTOM, 27-19
 - CCM_CUSTOM_NOW, 27-20
 - CCM_FORMSIZED, 27-12
 - CCM_GETFLAGS, 27-13
 - CCM_GETNUM, 27-14
 - CCM_GETSTR, 27-16
 - CCM_GETSTRCOUNT, 27-15
 - CCM_GETSTRLEN, 27-16
 - CCM_QDEFAULTSIZE, 27-12
 - CCM_RESETCONTENT, 27-12
 - CCM_SETNUM, 27-14
 - CCM_SETSTR, 27-17
 - WM_CREATE, 27-10
 - WM_DESTROY, 27-10
 - WM_ENABLE, 27-11
 - WM_GETTEXT, 27-11
 - WM_GETTEXTLENGTH, 27-11

- WM_QUERYWINDOWPARAMS, 27-11

- WM_SETFONT, 27-11

- WM_SETTEXT, 27-11

- WM_SETWINDOWPARAMS, 27-11

- numeric value messages, 27-13 to 27-15
 - options, 27-8

- OS/2 messages, 27-10 to 27-11

- overview, 27-1 to 27-2

- posting a message to the form procedure, 27-19

- query messages, 27-12 to 27-13

- sending messages to CA-Realizer, 27-19 to 27-20

- string value messages, 27-15 to 27-17

- style flags, 27-20

- Windows messages, 27-10 to 27-11

CUSTOMCONTROLS, 27-3

CVDMBF, 4-20

CVI\$, 4-20

CVL\$, 4-20

CVS\$, 4-20

CVSMBF, 4-20

D

data structures, nested complex, 2-12

Data types

- advanced, 2-3, 2-5

- alpha, 2-2, 2-16

- complex, 2-10

- date-time, 2-2, 2-9, 2-16

- definition of, 2-1

- introduction to, 2-1

- numeric, 2-2, 2-5

- real, 2-16

- reals, 2-7

- Simple, 2-2

- simple, 2-5

- Simple vs. Complex, 2-2
- string, 2-2, 2-8
- user-defined, 2-21 to 2-24

DATA, equivalent in CA-Realizer, 2-30

Data-relative position, converting to from screen-relative position, 11-54

Date-time

- data type, 2-2, 2-9, 2-16
- functions, 3-29 to 3-36
- operators, 3-3
- range, 2-9
- setting the system date and time, 3-33
- values
 - creating and converting, 3-31
 - day of week, 3-33
 - examining, 3-32 to 3-35
 - formats, 8-27
 - information, 3-33 to 3-35
 - manipulation of, 3-35 to 3-36
 - reading from a file, 7-18

DATE\$, 3-33

DateInfo, 3-5, 3-30, 3-34

DateToNum, 3-5, 3-30, 3-32, 3-33

DayOfWeek, 3-30, 3-33

DBF file, see XBase

DDE, 25-1 to 25-19

- _AllFormats, 25-5
- _Execute, 25-9
- advising, 25-8
- application name, 25-3, 25-4
- checking message queue, 25-10
- client
 - application, 25-2
 - messages, 25-8 to 25-9
 - sample session, 25-11 to 25-13
- closing a session, 25-11, 25-16
- data formats, 25-4, 25-5 to 25-8
- exchanging data, 25-8
- executing commands, 25-9
- how it works, 25-2
- initiating a client session, 25-3 to 25-5

- messages
 - from a server, 25-9 to 25-10
 - sending to server, 25-8 to 25-9
- OS/2 data formats, 25-6 to 25-8
- overview, 25-1 to 25-3
- poking data to a server, 25-9
- procedure, 25-9 to 25-10
- requesting
 - advisement from a server, 25-8
 - commands to be executed, 25-9
 - data from a server, 25-8
- responding to messages from a server, 25-9 to 25-10
- sample client session, 25-11 to 25-13
- selecting a session, 25-5
- sending
 - data to a server, 25-9
 - messages to a server, 25-8 to 25-9
 - server messages to a client, 25-16
- server
 - CA-Realizer, 25-13 to 25-16
 - messages, 25-9 to 25-10
 - overview, 25-2
 - session, 25-13
- session
 - information, 25-16 to 25-17
 - number, 25-3
 - overview, 25-2
- specifying
 - application name, 25-3
 - topic, 25-3
- title, 25-4
- topics, 25-3, 25-4
- unadvising, 25-9
- using SHELL to launch application, 25-4
- waiting for a message, 25-10
- Windows data formats, 25-5 to 25-6

DDEControl, 25-8 to 25-9, 25-11, 25-16

DDENew, 25-3 to 25-5, 25-14

DDEQ, 25-5, 25-16 to 25-17

DDEQUnique, 25-3

DDESelect, 25-5

DDESetProc, 25-9 to 25-10

DDEWait, 25-10

Declaring
 data types, 2-23
 variable types, 2-21

Default
 PRINT formats, 21-10 to 21-11
 Print Log, 8-17

Deleting and inserting portions of a string,
4-15

Dialog boxes, standard CA-Realizer, 8-1

DIM
 declaring
 a record type, 2-47
 an array, 2-33
 an array of records, 2-48
 explicit declaration of variables, 2-22
 statement, 2-22

Directory
 generating a file list using the FILES
 command, 7-8
 manipulation commands and functions,
 7-7

Displaying
 information in a log, 14-3
 messages
 with the Input dialog box, 8-13
 with the Message dialog box, 8-14

DLL, 26-1 to 26-2
 custom control, 27-2 to 27-3
 External function, 26-1
 External procedure, 26-1
 OS/2 functions, 26-11 to 26-12
 Windows functions, 26-11 to 26-12

DOS
 synchronization with CA-Realizer, 22-5
 system commands, 22-1 to 22-5

Double-precision real number, 2-7

Drag and drop, 15-75 to 15-78

Drawing on a tablet, see Graphics tablet

Dynamic array expansion, 2-33

Dynamic Data Exchange, see DDE

Dynamic Link Library, see DLL

E

E notation, 2-7

ELSE, see IF

ELSEIF, see IF

END FUNC, 6-5

END PROC, 6-3

EndValid, 2-18, 3-20, 3-22, 4-12

Eqv, 3-7

ERF, 5-19

ERL, 5-19

ERR, 5-19

Error handling, 5-18 to 5-20
 using
 the ERL function, 5-19
 the ERR function, 5-19
 writing a routine, 5-19 to 5-20

Error log, 21-9

Error trapping
 turning on and off, 5-18
 with ON ERROR, 5-18 to 5-20

Excel
 exporting and importing data from
 using the Named format, 7-30
 files, 7-31

Exchanging
 data using DDE, 25-8
 values, 2-19

EXECUTE
 checking for keywords using the
 SetKeyWord command, 6-25
 command, 6-24

EXIT FOR, 5-14

EXIT FUNC, 6-3

EXIT IF, 5-9

EXIT LOOP, 5-13, 5-14

EXIT MACRO, 5-19

EXIT PROC, 6-3

EXIT SELECT, 5-9

EXIT WHILE, 5-14

Exiting a procedure or function, 6-2 to 6-4

Exp, 3-15

Exp10, 3-16

Explicit declaration
 of a record type using DIM, 2-47
 of an array of records using DIM, 2-48
 of fixed-length string variables, 2-22
 of variables, 2-22

Exponential
 functions, 3-15
 notation, 2-7

Exporting and importing data
 CA-Realizer Named format, 7-28
 Excel and Lotus 1-2-3 using the Named
 format, 7-30
 storing and retrieving sets of variables
 in a file, 7-24
 text using the Named format, 7-31
 XBase files, 7-34 to 7-42

Expressions
 definition of, 3-1
 introduction to, 3-1 to 3-2
 overview, 3-9
 subscripting, 3-14

EXTERNAL, 26-2 to 26-3, 27-2 to 27-3

External function, see External procedure

External procedure
 declaring, 26-2 to 26-3
 functions, 26-10 to 26-11
 parameter types
 character, 26-6 to 26-7
 numeric, 26-3 to 26-6
 pointer, 26-8 to 26-10
 structure, 26-7 to 26-8

F

Families
 adding family names to a list box, 15-32
 arrays of, 2-53
 data types, 2-11
 members, 2-52
 in nested complex data structures, 2-54

FileClose, 7-8 to 7-22

FileDB, 7-23, 7-25, 7-34 to 7-42

FileEOF, 7-22

FileExport, 7-22 to 7-34

FileImport, 7-22 to 7-34

FileOpen, 7-8 to 7-22

FilePos, 7-10

FileQ, 7-9

FileQUnique, 7-11

FileRead, 7-8 to 7-22

Files

- adding names to a list box, 15-32
- Binary format, 7-34
- CA-Compete! files, 7-31
- CA-SuperCalc files, 7-31
- closing with FileClose, 7-15
- concepts in CA-Realizer BASIC, 7-4
- determining size of with LOF, 7-10
- Excel files, 7-31
- importing and exporting
 - data, 7-28
 - text, 7-33
- Lotus 1-2-3 files, 7-31
- manipulation using high-level commands, 7-22 to 7-34
- Named format, 7-26
- naming convention, 7-5
- Plain format, 7-33 to 7-34
- pointer
 - definition of, 7-10, 7-20
 - determining the position of, 7-20
 - positioning, 7-21
 - repositioning of, 7-21
- querying using FileQ, 7-9
- reading
 - an array of strings from, 7-18
 - date-time values, 7-18
 - numeric values, 7-18
 - using FileRead, 7-15
- Realizer format, 7-29 to 7-30
- search path, 7-5, 7-6
- spreadsheet files, 7-31
- writing multiple lines, 7-12

FILES command, 7-8

FileSeek, 7-10

FileWrite, 7-8 to 7-22, 7-12

Finding and replacing text within strings, 4-11 to 4-13

Fix, 3-17

Fixed-length string, 2-8

Floor, 3-18

FontControl, 10-5

FontNew, 8-9, 10-2, 10-4

FontQ, 10-5

FontQUnique, 10-3

Fonts

- adding font sizes to a list box, 15-33
- commands and functions, 10-1
- creating, 10-2
- destroying, 10-5
- dialog box, 8-9
- manipulating, 10-4
- querying with FontQ, 10-5
- selecting
 - from the standard font dialog box, 10-4
 - the current font, 10-4, 12-8
- setting
 - in a chart, 11-45
 - in a spreadsheet, 12-8
- undefining a font, 10-5

FontSelect, 10-4, 12-8

FOR...NEXT, 5-11

- general form, 5-11
- STEP keyword, 5-12
- TO keyword, 5-12

Form

- accelerators, 15-73 to 15-75
- controlling, 15-14 to 15-16
- drag and drop, 15-75 to 15-78
- modal, 15-54
- modeless, 15-54
- processing, 15-54

Formal parameter, 6-9

- defined, 6-9
- mixing with variables in expressions, 6-10 to 6-11
- referencing of, 6-8 to 6-9
- rules for usage, 6-10

Formats

- date-time, 8-27
- DDE data, 25-5 to 25-8
- defaults, 21-10 to 21-11
- fractional, 8-24
- mixing, 8-27
- numeric, 8-20
- string, 8-19

FormControl, 15-12, 15-14 to 15-16

FormDev, adding custom controls with, 27-5

FormModifyObject, 15-38 to 15-41, 27-15

FormQNum, 15-47, 27-14

FormQObject, 27-14

FormQStr, 15-48, 27-15

Forms, 15-1

- _Bitmap, 15-30
- _Metafile, 15-30
- accelerator key in, 15-23
- adding
 - bitmap button, 15-30
 - bitmap check box, 15-27
 - bitmap option button, 15-26
 - caption, 15-29
 - check box, 15-27
 - clock, 15-36
 - combo box, 15-31
 - default button, 15-25
 - digital clock, 15-36
 - drop-down combo box, 15-31
 - drop-down list box, 15-31
 - edit controls, 15-28, 15-29
 - event timer, 15-37
 - frame, 15-34
 - group box, 15-28
 - interface elements and tools, 15-2 to 15-3
 - list box, 15-31
 - OLE object, 15-37
 - option button, 15-25, 15-26

PCX file, 15-30

picture, 15-30

scroll bar, 15-34

timer, 15-37

background color in, 15-19

check box, 15-44

closing

- form objects, 15-41

- using an accelerator key, 15-12

- using the System menu, 15-13

commands and functions, 15-2

controlling appearance of, 15-14

custom control, 27-1 to 27-21

determining text value of an object, 15-48

display independence, 15-38

edit control verification, 15-59, 15-68 to 15-69

Enter ID, 15-20

Escape ID, 15-20

event timer, 15-45, 15-57

field color in, 15-19

focus, 15-42

front-to-back ordering, 15-42 to 15-43, 15-47

graying an object, 15-42

interacting with, 15-66

list box, 15-45

- adding family names, 15-32

- adding file names, 15-32

- adding font sizes, 15-33

- adding item, 15-44

- adding variable names, 15-32

- deleting item, 15-44

- inserting item, 15-45

- selecting item, 15-44

manipulating an object in, 15-38 to 15-41

MDI windows, 15-50

messages, 15-55 to 15-60, 15-67 to 15-68

modeless, 15-65

non-MDI windows, 15-50

notification

- animation, 15-58
- bitmap, 15-57
- button, 15-55
- caption, 15-56
- check box, 15-56
- edit control, 15-56
- Enter, 15-58
- Esc, 15-58
- frame, 15-57
- group box, 15-56
- list box, 15-57
- option button, 15-56
- picture, 15-57
- scroll bar, 15-58
- tablet, 15-58
- timer, 15-57
- tools, 15-58

numeric value of an object, 15-47

object notification, 15-55 to 15-60

objects

- bitmap, 15-6
- bitmap button, 15-6
- button, 15-3
- caption, 15-6
- check bitmap, 15-5
- check box, 15-4, 15-5
- combo box, 15-7
- default button, 15-3
- drop-down combo box, 15-8
- drop-down list box, 15-7
- edit controls, 15-5
- event timer, 15-9
- frame, 15-8
- group box, 15-5
- list box
 - overview, 15-7
- metafile, 15-6
- OLE, 15-9
- option bitmap, 15-4
- option button, 15-4
- overview, 15-3
- PCX file, 15-6

scroll bar, 15-8

state, 15-41

text value, 15-48

timer, 15-9

option button, 15-44

palette, 15-52

Pick mode, 15-59 to 15-60, 15-69

PickDrag mode, 15-59 to 15-60, 15-69

popup, 15-53

positioning, 15-38

procedure, 9-21, 15-66

removing object, 15-41

scroll bar, 15-45

setting

- text color, 15-19

- the background grid in, 15-72

- the color of, 15-18 to 15-20

status bar, 15-51 to 15-52

timer, 15-45

toolbar, 15-51 to 15-52

tools, see Programmable Application

Tool, 15-38

unique object number in, 15-20

FormSelect, 15-13

FormSetColor, 15-18 to 15-20

FormSetGrid, 15-72

FormSetObject, 9-9, 10-6, 15-2 to 15-3,
27-3 to 27-10, 27-15

FormSetProc, 15-65

Fractional formats, 8-24

Frame, 15-8

FUNC...END FUNC, 6-4 to 6-7

Functions, 3-10 to 3-36

- date-time, 3-29 to 3-36

- definition of, 3-2

- exiting, 6-5

- exponential, 3-15 to 3-17

- expression evaluation within, 6-21

- external, 26-10 to 26-11

- introduction to, 3-1 to 3-2

- manipulating random numbers,
3-18 to 3-19
- matrix manipulation, 3-23 to 3-24
- parameters for, 3-10
- rounding numbers, 3-17 to 3-18
- statistical, 3-24 to 3-26
- time series, 3-27 to 3-29
- trigonometric, 3-15 to 3-17

Functions and procedures, see Procedures
and functions

G

Gantt charts, see Charts

Global variables defined, 6-14

GOTO label, 5-9

Graphics tablet

- adding text, 16-15
- buffering modes, 16-17
 - _Bitmap, 16-17
 - _BufferedMetafile, 16-17
 - _Metafile, 16-17
- controlling the redraw, 16-18 to 16-19
- creating, 16-2
- drawing
 - a line in, 16-7
 - commands, 16-4 to 16-15
 - complex shapes, 16-12
 - program example, 16-23
 - simple shapes, 16-9
- manipulating
 - bitmaps, 16-13
 - pixels, 16-13
- mouse click processing, 16-16
- partial update, 16-19 to 16-20
- positional notation, 16-2
- processing in using a form procedure
or FormWait, 16-16
- querying the color of an individual
pixel, 16-14

- selecting, 16-3
- Tic Tac Toe, 16-21
- undoing commands, 16-19 to 16-20
- VGA and Super VGA, 16-2

H

Help, 1-10, 20-1 to 20-4

- application specific, 20-1 to 20-4
- customizing menu, 20-2
- index, 20-2
- Manager, 20-1 to 20-3
- overriding default, 20-4
- procedure, 20-3 to 20-4

HelpSetProc, 20-3 to 20-4

Hex\$, 4-16

Hexadecimal notation, 2-6

HexToNum, 4-18

High-level file commands and functions

- creating and manipulating files, 7-4
- guidelines for usage, 7-2
- Named file format, 7-26

Horizontal bar chart, see Charts

I

IEEE floating point standard, 2-7

IF, 5-4 to 5-7

- ELSE, 5-4 to 5-7
- ELSEIF, 5-6
- general form for usage, 5-4
- THEN, 5-4
- used as a function, 5-7
- with the ELSE keyword, 5-5

Imp, 3-7

Importing and exporting data, see
Exporting and importing data

Index function, 2-30, 3-20, 3-21, 3-33

Initiating DDE conversations, 25-3

INPUT, 8-11

Inserting and deleting portions of a string,
4-15

Installing Windows under OS/2, 1-6

InStr, 4-8, 4-9, 4-12

Integer
 converting to a string, 4-16
 definition of, 2-5, 2-6
 notation used, 2-6

Integrating a CA-Realizer tool into an
application, 9-2

Invoking a log procedure, 14-12

IS, see SELECT

Iterative statement
 definition of, 5-10
 guidelines for usage, 5-10
 introduction to, 5-4, 5-10
 LOOP...END LOOP, 5-12 to 5-13
 optimizing with arrays, 5-14 to 5-18
 types in CA-Realizer, 5-10

K

Keywords

 CA-Realizer reserved, 2-14
 checking for CA-Realizer reserved
 keywords, 6-25

L

Launching other applications, 22-1 to 22-3

LCase\$, 4-14

Left\$, 4-6

Len, 4-3, 4-9

LFLUSH, 9-20, 11-57

Libraries
 custom control, 27-2 to 27-3, 27-5
 dynamic link, 26-1 to 26-2

Line chart, see Charts

Line wrapping length, 21-11

LOCAL, 6-14

Local variable
 declared using LOCAL, 6-14
 definition of, 6-14

LOF, 7-10

Log function, 3-15

Log tool
 adding text, 14-4
 changing characteristics of information
 in, 14-9 to 14-11
 character, 14-8
 clearing, 14-5
 command summary, 14-1
 controlling appearance of, 14-4 to 14-7
 creating, 14-1 to 14-2
 customizing with a log procedure,
 14-12
 extracting data, 14-8
 ID number, 14-2
 invoking a log procedure,
 14-11 to 14-13
 limiting text, 14-10
 line numbering, 14-8
 position, 14-8
 preventing close, 14-13
 printing to, 14-3, 14-4

- procedure, 9-21, 14-12
- processing in a log procedure, 14-13
- querying a log, 14-7 to 14-9
- specifying
 - a log number, 14-3
 - tab stops, 14-7
 - the font, 14-6
- Log10, 3-16
- LogControl, 14-4 to 14-7
- Logical operators, 3-5 to 3-7
 - creating logical statements, 3-6
 - defined, 3-6
- LogNew, 14-1 to 14-2
- LogQ, 14-7 to 14-9
- LogQData, 14-8
- LogQUnique, 14-2
- LogSetData, 14-9 to 14-11
- LogSetProc, 14-11 to 14-13, 14-12
- LOOP...END LOOP, 5-10, 5-12 to 5-13
- Loops, optimizing by using array processing, 5-14 to 5-18
- Lotus 1-2-3
 - exporting and importing data from using the Named format, 7-30
 - files, 7-31
- Low-level file commands and functions
 - checking for EOF, 7-22
 - closing a file, 7-15
 - creating and manipulating files, 7-4
 - determining the position of the file pointer, 7-20
 - for opening files, 7-11
 - for writing to a file, 7-12
 - guidelines for usage, 7-2
 - positioning the file pointer, 7-12

- reading
 - a line from a file, 7-16
 - an array of strings from a file, 7-18
 - from a file, 7-15
 - multiple lines from a file, 7-17
 - portions of a string from a file, 7-17
- repositioning the file pointer, 7-20
- writing
 - a line of text to a file, 7-13
 - an array of strings to a file, 7-14
 - numeric and date-time values to a file, 7-15

LPRINT, 8-16, 9-20

LROWPRINT, 8-16, 9-20

LTrim\$, 4-15

M

Macro, 6-23

Manipulating

- a font, 10-4
- a picture, 10-9
- date-time values, 3-35 to 3-36
- directories, 7-7
- strings, 4-1 to 4-21

Marker chart, see Charts

Mask, 3-20

Mathematical

- errors
 - floating point, 21-13
 - indefinite value, 3-30
 - infinite value, 3-30
 - not a number, 3-30
 - reporting of, 3-30
 - stopping execution and displaying, 3-30
- operators, 3-3

MatInvert, 3-23

MatMult, 3-23

MATRANSPOSE, 2-32

Matrix

- functions, 3-23 to 3-24
 - MatInvert, 3-23
 - MatMult, 3-24
 - MatTranspose, 3-23
- transposing, 2-32

Max, 3-20, 3-22

MDI

- region, 15-50
- resizing window, 15-51, 21-7
- windows, 15-50

member names, listing, 13-6

MenuControl, 18-6, 18-7, 18-8

MenuDoCmd, 18-17

MenuNew, 18-3

MenuQUnique, 18-3

Menus

- adding
 - a menu item to the current menu, 18-10
 - a menu to the menu bar, 18-7
 - a separator between items, 18-11
- adjusting the position on the menu bar, 18-7
- changing item text, 18-12
- checked state, 18-12
- closing, 18-9
- controlling
 - redraw, 18-9
 - the appearance of, 18-6 to 18-10
- creating, 18-3
 - a submenu, 18-19
 - with MenuNew command, 18-3 to 18-16
- definition of, 18-1
- grayed state, 18-13
- graying item, 18-13
- hiding, 18-8
- incorporating a CA-Realizer system menu command, 18-17

menu bar, 18-1

modifying a menu item, 18-12

normal state, 18-13

procedure, 9-21, 18-14

redisplaying, 18-8

removing

- a menu item, 18-14

- from the menu bar, 18-6

replacing a CA-Realizer system menu, 18-8

selecting the current menu, 18-5

simulating the action of a CA-Realizer system menu, 18-17

specifying

- a single key accelerator for a menu item, 18-21

- an accelerator key for, 18-4

- an ALT key accelerator for a menu item, 18-21

submenu, 18-2

unique identification number for a menu item, 18-10

MenuSelect, 18-5

MenuSetCmd, 18-10, 18-17

MenuSetProc, 18-14

Messages, notification, 15-55

Mid\$, 4-6, 4-10

Min, 3-20, 3-22

Mixing formats, 8-27

MKDIR, 7-7

MKDMBF\$, 4-20

MKI\$, 4-20

MKL\$, 4-20

MKS\$, 4-20

MKSMBF\$, 4-20

Modifiers used in a procedure or function, 6-16

Modifying a form object, 15-38 to 15-41

Modules, 6-22 to 6-24

Morphing arrays, 2-35 to 2-37

Moving

- a pointer in a file, 7-21

- averages, 3-28

- to a cell in a sheet, 12-13

Multi-dimensional array, 2-25

N

Named file format, 7-26

Naming conventions

- files, 7-5

- variables, 2-14

Nested control structures, 5-17

Non-MDI window, 15-13

Not, 3-6

Notation, exponential, 2-7

Notification messages, 15-55

Numeric

- converting to a string, 4-16

- data type, 2-2

- formats, 8-20

- reading values from a file, 7-18

- variable type, 2-16

NumToDate, 3-31, 3-32, 3-34

NumToStr, 4-16

O

Oct\$, 4-16

Octal notation, 2-6

OctToNum, 4-18

ON ERROR, 5-4, 5-18 to 5-20

ON ERROR GOTO

- error handling, 5-21

- label, 5-19

- statement, 5-18

Opening files, see Low-level file commands

Operating on arrays, 2-37 to 2-44

Operating system commands, 22-1 to 22-5

Operators

- bitwise, 3-7 to 3-8

- comparison, 3-4 to 3-5

- concatenating using, 3-4

- date-time, 3-3

- definition of, 3-1

- introduction to, 3-1 to 3-2

- logical, 3-5 to 3-7

- mathematical, 3-2 to 3-4, 3-3

- rules of precedence, 3-9, 3-8 to 3-10

Optimizing iteration by using arrays,
5-14 to 5-18

Optional modifiers

- determining

 - the number of, 6-18

 - the value of, 6-19

- in procedures and functions,
6-17 to 6-21

- querying

 - with QNOptMods, 6-18

 - with QOptMod, 6-18

Optional parameters

- determining
 - the number of, 6-18
 - the value of, 6-19
- in procedures and functions, 6-17 to 6-21
- querying
 - with QNOptParams, 6-18
 - with QOptParam, 6-18

Or, 3-6

OS/2

- accessing system DLL functions, 26-11 to 26-12
- DDE data formats, 25-6 to 25-8
- system commands, 22-1 to 22-5

P

Packed

- end valid (PEV), 3-26
- start valid (PSV), 3-26

Packing

- arrays, 3-26
- sparse arrays, 3-26

Palette

- creating, 15-52
- defining, 15-52

Panes, 11-49 to 11-51

Parameters

- actual, 6-9
- definition of, 6-8
- explicit declaration of, 6-8
- formal, 6-8 to 6-9
- general form for usage, 6-8
- passing
 - by reference, 6-11
 - by value, 6-11
- rules for using actual and formal, 6-10

Params, 9-23 to 9-34

Passing parameters

- by reference, 6-11 to 6-13
- by value, 6-11 to 6-13

Pasting

- a picture to the clipboard, 23-2
- text into the Clipboard, 23-1

Path, 7-5

Pens and Brushes, 16-5

PEV (packed end valid), 3-26

PictureClipGet, 10-6, 10-10, 23-2

PictureClipSet, 10-11, 23-2

PictureControl, 10-10

PictureNew, 10-6, 10-7, 10-9

PictureQ, 10-9

PictureQUnique, 10-6

Pictures

- bitmap, 10-6
- copying
 - from the Clipboard, 10-6, 10-10
 - to the Clipboard, 10-11
- creating, 10-6
- destroying, 10-10
- manipulating, 10-9
- metafile, 10-6
- modifying and manipulating, 10-9
- querying with PictureQ, 10-9
- selecting the current picture, 10-9
- tool
 - commands, 10-1
 - functions, 10-1
- using with the Animator, 17-9 to 17-12

PictureSelect, 10-9

Pie charts, see Charts

Pixels, 9-15

Plain format, 7-33 to 7-34

Plotting in a chart, 11-48

Pointer, 26-9

Points, 9-15

Position, converting from screen-relative position to data-relative position, 11-54

Positional notation

- converting between inches, points, twips, and millimeters, 9-18
- in a graphics tablet, 16-2
- positioning a tool, 9-14
- units, 9-15

Positioning

- pointer in a file, 7-21
- programmable application tool on screen, 9-14, 9-15

Precedence of operators, 3-9

Predefined constants, 2-20

PRINT, 4-20, 8-16, 14-3, 14-4

Print log, 14-1 to 14-14, 21-9

Printer

- configuration, 8-6
- controlling programmatically, 8-7
- improving printer quality, 21-14

PrinterControl, 8-7

Printing

- arrays, 2-44, 8-17
- date-times, 3-32
- determining line length, 21-11
- families, 8-17
- flushing the buffer with LFLUSH, 8-30
- programmable application tools, 9-19
- to logs, 8-17, 14-3, 14-4

PROC...END PROC

- determining when to use, 6-2
- using to declare a procedure, 6-2 to 6-4

Procedures

- application, 9-32
- DDE, 25-9 to 25-10
- exiting, 6-3
- external, see External procedure
- function and expression evaluation, 6-21
- help, 20-3 to 20-4
- passing
 - by reference, 6-11
 - by value, 6-11

Procedures and functions

- as replacements for GOTO and GOSUB, 6-1
- definition of, 6-1
- guidelines for usage, 6-2
- invoking, 6-2
- modifiers, 6-15 to 6-17
- optional parameters and modifiers, 6-17 to 6-21
- passing
 - a constant, 6-11 to 6-13
 - a variable, 6-11 to 6-13
 - an expression, 6-11 to 6-13
 - parameters, 6-7 to 6-11
- recursion, 6-21 to 6-22
- redefining, 6-21
- used with EXIT, 6-4

Processing

- in the chart procedure, 11-54
- in the log procedure, 14-13
- in the sheet procedure, 12-23

Product function, 3-20

Programmable Application Tools, 9-2

- allocating unique identification numbers, 9-4
- animator, 17-1
- boardwatch, 13-1 to 13-15
- categories, 9-2 to 9-3
- chart, 11-1 to 11-55
- closing, 9-20
- command summary, 9-34 to 9-38
- common commands, 9-3
- controlling state, size, and location, 9-12

- creating a new tool, 9-6
- displaying, 9-13
- hiding, 9-12
- incorporating fonts and pictures, 10-1
- log, 14-1 to 14-14
- maximizing, 9-12
- minimizing, 9-12
- parameters for tool procedures, 9-23 to 9-34
- positioning on the screen, 9-14
- printing, 9-19
- Scheduler, 19-1
- selecting, 9-10
- showing, 9-13
- sizing of, 9-14
- specifying a tool procedure, 9-21
- spreadsheet, 12-1 to 12-25
- tool procedure
 - parameter array, 9-23 to 9-34
 - setting, 9-21
- using in a CA-Realizer application, 9-2
- window style, 9-6

Programming

- considerations, 2-1
- controlling the flow of a program, 5-2
- data types, 2-1

PSV (packed start valid), 3-26

Q

QDate, 3-10, 3-31, 3-33

QDir, 7-7

QNOptMods, 6-18

QNOptParams, 6-18

QOptMod, 6-19

QOptParam, 6-19

Querying

- a file on disk, 7-9
- an XBase file, 7-39
- the Sheet Selection, 12-20

QVar, 2-18, 12-23, 13-14

R

Random

- access files, 7-1
- number functions, 3-18 to 3-19
- number generator
 - reseeding, 3-19
 - uniform, 3-19

Randomize, 3-19

Ranges

- date-time, 2-9
- setting for rows and columns, 12-11

Reading

- from a file, 7-15
- numeric and date-time values, 7-18

Real number

- converting to a string, 4-16
- definition of, 2-5
- double precision, 2-7
- notation used, 2-7
- single precision, 2-7

Realizer data format, 7-29 to 7-30

Records, 2-11

- accessing fields, 2-48
- allocating arrays of records, 2-48
- data type, 2-45
- extracting a column, 2-50
- in nested complex data structures, 2-49

Recursion, 6-21 to 6-22

Redefining procedures and functions, 6-21

REDIM, the equivalent of in CA-Realizer, 2-33

Removing directories, 7-7

Replacing and finding text within strings, 4-11 to 4-13

Reporting mathematical errors, 3-30

Reserved words, 2-14

RESET, 8-29, 15-12, 18-6

Resetting CA-Realizer, 15-12

RESUME, 5-20 to 5-21

RESUME...NEXT, 5-19

Retrieving parts of a string, 4-5 to 4-11

RETURN, see FUNC...END FUNC

Reverse\$, 4-3

RGB color, determining value with QSys, 11-22

Right\$, 4-6, 4-9

RMDIR, 7-7

Rnd, 2-33

Round, 3-17

Rounding functions, 3-17 to 3-18

- Rnd, 3-19
- Round, 3-18

ROWPRINT, 8-16

RTrim\$, 4-15

RUN, 6-22 to 6-24

RunTot, 3-20

S

Saving a file, 8-4

Scalars, rules for computation, 3-13

SchedQData, 19-10

SchedRemove, 19-4, 19-9

SchedSet, 19-4

Scheduled event, see Scheduler

Scheduler

- adding
 - a periodic event, 19-7
 - an event, 19-5
- anatomy, 19-2
- clearing, 19-9
- event in, 19-4
- removing
 - a periodic event, 19-9
 - event, 19-3
- running a scheduled command, 19-4
- scheduling an event, 19-4
- showing, 19-2
- start button, 19-3
- starting, 19-6
- stop button, 19-3
- storing and executing events, 19-1

Scoping

- global, 6-14
- local, 6-14
- variables, 6-13 to 6-15

Screen-relative position, converting to data-relative position, 11-54

Search path, 7-5

Searching directories, 7-5

SELECT

- conditional statement, 5-4
- definition of, 5-8
- evaluation of statement value range, 5-8
- function, 6-6
- general form for usage, 5-8
- using with the IS keyword, 5-9

Selecting the current Programmable Application Tool, 9-10

Sequential files, 7-1

Serial communications

- baud rate, 24-2
- buffer size, 24-2
- byte size, 24-2
- closing a port, 24-6
- commands, 24-1
- opening a port, 24-1 to 24-3
- parity, 24-2
- querying a port, 24-4 to 24-6
- reading from a port, 24-3
- stop bits, 24-2
- writing to a port, 24-3 to 24-4

Serial ports, See Serial communications

SetDir, 7-7

SetFormat

- date-time values, 7-33
- default format, 12-12
- real values, 7-33
- specifying default format, 8-29
- using to assign a format to a variable in a board, 13-8
- using to assign a format to a variable in a spreadsheet, 12-12

SetSys, 3-5, 7-33

- _AlphaFormat, 8-28
- _CaseSensitive, 3-5, 4-14
- _DateFormat, 8-28
- _FPError, 3-30
- _LoadDir, 7-5

_MacroDir, 7-5

_ProgDir, 7-5

_RealFormat, 8-28

_Separator, 7-32

system variables, 21-1

SetSystemDate, 3-33, 22-6

SetSystemTime, 3-33, 22-6

Setting

- colors for column headings, 12-7
- colors for row headings, 12-7
- colors for sheet cells, 12-7
- fonts for column headings, 12-8
- fonts for row headings, 12-8
- fonts for sheet cells, 12-8
- ranges for rows and columns, 12-11
- system date and time, 3-33
- the default format on a Sheet, 12-12
- the log font, see Log tool
- the system date and time, 22-6

Sgn, 3-17

Sheet, see Spreadsheets

Sheet procedure, 9-21, 12-22

SheetControl, 12-3, 12-11, 12-13

SheetControlLabels, 12-16

SheetNew, 12-3

SheetQInfo, 12-20

SheetQUnique, 12-3

SheetSelect, 12-3

SheetSetProc, 12-22

SheetSetRowLabels, 12-14

SheetUpdate, 12-13, 12-17, 15-38

SHELL, 22-1 to 22-3, 22-4

Shifting elements in arrays, 3-13

Shortcut key, see Accelerator key

Sin, 3-15

Single-precision real number, 2-7

Sizing

- a Programmable Application Tool, 9-14
- CA-Realizer window, 21-6

Sorting an array with StatSort, 3-25

Space\$, 4-3

Sparse arrays, 3-26

Specifying

- a board template, 13-5
- application name in DDE, 25-3
- log numbers, 14-3
- the chart type, 11-7
- topics in DDE, 25-3

Spreadsheets, 12-1 to 12-25

- anatomy, 12-2
- arrays, 12-4
- attaching a sheet procedure, 12-22
- clearing variable, 12-18
- column
 - buttons, 12-10
 - headers, 12-14
 - width, 12-17
- command summary, 12-1
- controlling the appearance of, 12-6
- creating, 12-3
- customizing with a sheet procedure, 12-22
- default formats for, 12-12
- families, 12-4
- files, 7-31
- headers, 12-14
- labels, 12-14
- processing in the sheet procedure, 12-23
- querying, 12-20
- read-only, 12-11
- resizing column width, 12-9
- row headers, 12-14
- scroll bars, 12-13

setting

- attribute to Read-Only, 12-18
- colors for cells, row headings, and column headings, 12-7
- fonts for cells, rows and column headings, 12-8
- sheet procedure, 12-21 to 12-24
- two-dimensional array, 12-4, 12-5
- updating the contents, 12-17

Sprint function, 3-32, 4-20

Sqr, 3-17

Sqrt, 3-15, 3-17

Standard dialog boxes, 8-1

StartValid, 2-18, 3-20, 3-22

Statements, 5-1

Statistics functions, 3-24 to 3-26

- StatMovAvg, 3-28
- StatNorm, 3-25
- StatPack, 3-26
- StatPercent, 3-25
- StatRank, 3-25
- StatSort, 3-24, 4-14
- StatSwitch, 3-27
- StatUnpack, 3-26, 3-27

StatSwitch, 3-27

Status bar, 15-51 to 15-52

StdColor, 8-10

StdOpen, 8-2

StdOpen Dialog box, 8-2

StdPrintConfig dialog box, 8-6

StdSaveAs dialog box, 8-4

STEP, see FOR...NEXT

StrDelete\$, 4-15

STRING * n data type, 2-3, 2-8

STRING data type, 2-3, 2-8

String\$, 4-3

Strings

- changing case of, 4-14
- comparing guidelines, 4-5
- comparison operators, 4-4 to 4-5
- concatenating, 4-4
- converting
 - from external formats, 4-19
 - numbers to, 4-16
 - to numbers and date-times, 4-17
- data type, 2-2, 2-8
- definition of, 2-8
- determining the length of using Len, 4-9
- finding and replacing text in, 4-11 to 4-13
- fixed-length, 2-8
- format codes, 4-21
- formats, 8-19
- formatting
 - with format codes, 4-21
 - with Sprint function, 4-20
- functions for retrieving parts of, 4-5 to 4-11
- in BASIC, 4-1
- inserting and deleting portions of, 4-15
- manipulating individual characters using Asc and Chr\$, 4-2
- retrieving
 - a part of using Instr, 4-7 to 4-11
 - a part of using Left\$, 4-6
 - a part using Mid\$, 4-6
 - a part using Right\$, 4-6
- size, 2-8
- terminating with carriage return and linefeed combination, 7-13
- tokenizing using StrTok, 4-7 to 4-11
- variable declaration, 2-16
- variable-length, 2-8

StrInsert\$, 4-15

StrToDate, 3-31, 3-32, 4-17, 19-6

StrToNum, 4-17, 8-9

SUB, 6-3

Sub-arrays, 2-44

Subarrays, 2-39 to 2-42

Submenu, 18-2

Subranging arrays, 2-30

Subscripting expressions, 3-13 to 3-14

SubStr\$, 4-12

Sum, 3-20, 3-21

SuperVGA, 9-16

SWAP, 2-19

swapping values, 2-19

Switching arrays, 3-26

Symbolic constants, 2-20

System

- setting date, 3-33

- setting time, 3-33

System variables

- command-specific, 21-11 to 21-13

- default PRINT formats, 21-10 to 21-11

- logs, 21-9 to 21-10

- paths, 21-7 to 21-9

- querying, 21-1

- size values, 21-5 to 21-7

- system constants, 21-13 to 21-14

- tuning, 21-1

T

Tablet, see Graphics tablet

TabletArc, 16-10

TabletBitmap, 16-13

TabletChord, 16-10

TabletControl, 16-17, 16-18

TabletEllipse, 16-10

TabletGetPixel, 16-14

TabletLine, 16-7

TabletLineTo, 16-9
TabletMouseX, 16-16
TabletMouseY, 16-16
TabletPie, 16-11
TabletPolygon, 16-12
TabletPolyline, 16-8
TabletPolyPolygon, 16-12
TabletRectangle, 16-9
TabletRoundRect, 16-9
TabletSelect, 16-3
TabletSetPixel, 16-9, 16-13
TabletText, 16-2, 16-15
TabletTextColor, 16-15
TabletUndo, 16-17, 16-19 to 16-20
Tan, 3-15
Text
 adding to the graphics tablet, 16-15
 converting to ASCII, 4-2 to 4-4
 edit controls, 15-5
Text Log tool, 14-1
THEN, see IF
Tic Tac Toe, 16-21
Time-series functions, 3-27 to 3-29
TIMES\$, 3-33
Time, setting for system, 3-33
TO, see FOR...NEXT
Tool procedures, 9-21
Toolbar, 15-51 to 15-52
Tools, placing in a form, 15-9
Topics, specifying in DDE, 25-3
Transposing a matrix, 2-32

Trigonometric functions, 3-15
True and false operators, 3-4
Twips, 9-15
TYPE
 creating data types, 2-23
 statement, 2-23
 usage in declaring records, 2-45
Types
 basic data types, 2-1
 declaration of, 2-23
 user-defined, 2-23
 variable, 2-16

U

UCASE\$, 4-14
Unconditional
 control of program flow, 5-9
 statement, GOTO label, 5-9
Unpacking sparse arrays, 3-26
Updating a spreadsheet, 12-17
User-defined data types, 2-23

V

Val, see StrToNum
Valid array range, 2-27
values
 exchanging, 2-19
 swapping, 2-19
Values, assigning to a variable, 2-14

variable type, changing, 2-17

Variable-length string, 2-8

Variables

- affect of the CLEAR command on, 2-22

- assigning a value, 2-14

- clearing

 - in a Boardwatch, 13-12

 - with Clear command, 2-19

- date-time, 2-16

- declaring types, 2-21

- definition of, 2-13

- explicit declaration of, 2-22

- global, definition of, 6-14

- in a list box, 15-32

- information, 2-18

- local, definition of, 6-14

- naming convention, 2-14

- numeric, 2-16

- of type REAL, 2-16

- records, 2-11

- retrieving from a file, 7-24

- scope of, 6-13 to 6-15

- storing in a file, 7-24

- strings, 2-16

- system, 21-1

- types, 2-16

- undefined, 2-16

- valid names, 2-14

Vector graphics, 16-1

VectSet, 3-20, 3-21

Version, determining with `_Version`, 21-13

VGA, 9-16

W

WEND, see WHILE...END WHILE

WHILE...END WHILE, 5-10, 5-13

Window handle, 21-14

Windows

- accessing system DLL functions,
26-11 to 26-12

- DDE data formats, 25-5 to 25-6

Writing

- an error-handler routine, 5-19 to 5-20

- to a file, 7-12

X

XBase

- importing and exporting data from,
7-34 to 7-42

- querying a file, 7-39

Xor, 3-7

XY Charts, see Charts

Z

ZeroFill, 3-20

